$\mathcal{L}$ - Fuzzy Structured Query Language

Evans Adjei
Department of Computer Science

Supervisor:
Dr. Michael Winter

Submitted in partial fulfillment of the requirements for the degree of
Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario, Canada

Abstract

Lattice valued fuzziness is more general than crispness or fuzziness based on the unit interval. In this work, we present a query language for a lattice based fuzzy database. We define a Lattice Fuzzy Structured Query Language ($\mathcal{L}$FSQL) taking its membership values from an arbitrary lattice $\mathcal{L}$. $\mathcal{L}$FSQL can handle, manage and represent crisp values, linear ordered membership degrees and also allows membership degrees from lattices with non-comparable values. This gives richer membership degrees, and hence makes $\mathcal{L}$FSQL more flexible than FSQL or SQL. In order to handle vagueness or imprecise information, every entry into an $\mathcal{L}$-fuzzy database is an $\mathcal{L}$-fuzzy set instead of crisp values. All of this makes $\mathcal{L}$FSQL an ideal query language to handle imprecise data where some factors are non-comparable. After defining the syntax of the language formally, we provide its semantics using $\mathcal{L}$-fuzzy sets and relations. The semantics can be used in future work to investigate concepts such as functional dependencies. Last but not least, we present a parser for $\mathcal{L}$FSQL implemented in Haskell.

Acknowledgment

I would like to express my profound gratitude and acknowledgment to my supervisor Prof. Michael Winter. Through his competence and passion in this research area, he has worked hand in hand with me right from the beginning till the end of this piece of work. His patience, friendly and approachable nature encouraged me to seek his advice on issues regarding this thesis. From uncountable questions and answers interactions I had with him, I was inspired and nurtured to complete this work. I will also extend my sincere appreciation to my supervisory committee members Prof. Brian Ross and Prof. Ke Qiu for their support and guidance. I will again thank all the faculty members at the Department of Computer Science for their support and encouragement throughout my masters program.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Data and information management have been part of human life from the time immemorial. In this contemporary world, good management of information is one of the intrinsic elements in the success of an organization. The uses and benefits of databases are not limited to only corporate bodies. The phone book on our phones which is a database, is a practical example of the uses and benefits of databases to individuals. A database which is the collection of organized data for easy access, managing and retrieval of information has overcome the bottleneck of the traditional filing system by providing easy and flexible access to information, data sharing, data integrity, information security and optimization of space [20].

The relational database model, developed by E. F. Codd [5], has been widely used in this modern world due to its ability to break the barriers of complexity and inflexibility of the network and hierarchical data model. Relational Database Management Systems (RDBMS) serve as an interface program between data in the database and the human operator [5]. RDBMS have the significant power to manage date efficiently than even the spreadsheet softwares [18].

Structured Query Language (SQL), which is the standard language used to interact with RDBMS, is one of the fountainheads of the success, and predominate for relational databases in the commercial world of database systems. RDBMS and SQL are based on crisp concepts which use all-inclusive or all-exclusive approach. They use a Boolean filter process to operate, which makes it incapable of representing, handling and managing imprecise or vague information. Uncertain information for example, using labels such as young, short, long, or old does not have clear or precise bound-

aries, as opposed to information that can easily be distinguished by "Yes" or "No" [9].

There is a great portion of this imprecise information in our daily activities. We model a database to keep records of such daily activities. Labeling a person as being young in terms of age, for instance, is ambiguous. In one context, one could conclude that 30 years old and below represents young persons, whilst someone will go beyond to consider 40 years old and below as being young.

In addition, to consider 20 years old and 40 years old persons as young on the same scale will be inappropriate. It will be appropriate to represent them using a scale of degrees of membership, indicating to what degree they belong to the set of young people. Zadeh [6, 32] introduced the concept of fuzziness representing a generalization, which goes beyond the classical set concept by mapping elements to the unit interval between 0 and 1 inclusively [9]. In other words, it is a generalization based on the degree of membership rather than an all inclusive or all exclusive approach. Fuzzy Relational Database Management Systems (FRDBMS) and Fuzzy Structured Query Language (FSQL) are based on the theory and concept of fuzziness. These FRDBMS and FSQL have made it possible to model this imprecise and vague kind of information in databases.

In optimization problems, it is more appropriate to generalize beyond the unit interval since some elements may be incomparable [9]. $\mathcal{L}$-Fuzziness developed by Goguen is a more generalized concept than the fixed unit interval fuzziness [4, 9]. In other words, lattice-valued fuzziness ($\mathcal{L}$-fuzziness) is a generalization of the fuzziness that uses arbitrary lattice values instead of the unit interval as membership degrees. For example, rating a car as good or bad, in one context a person who likes fast moving cars will consider the horsepower to judge, but in another context, someone will consider the price to judge. Such situation rating is not linearly ordered and it is more appropriate to treat it in $\mathcal{L}$-fuzzy concept due to the property of partial order.

The purpose of this thesis is to develop $\mathcal{L}$-Fuzzy Structured Query Language ($\mathcal{L}$-FSQL) for FRDBMS and to give the semantics of $\mathcal{L}$FSQL using the abstract theory of arrow categories for $\mathcal{L}$-fuzzy. The motivation of this work is based on the fact that $\mathcal{L}$-fuzziness is more generalized than the Fuzzy concept by Zadeh as explained in this section. This $\mathcal{L}$FSQL will take its membership values from an arbitrary lattice. This is more generalized than FSQL which takes its membership values from a fixed unit interval. This in effect will give varieties of membership values. There are a lot of

essential operations and properties that come with lattices [14]. The properties, laws and operations satisfied in the lattice can be extended to the $\mathcal{L}$-fuzzy relation which gives indications that these essential properties, laws and operations can be used in $\mathcal{L}$FSQL as well. Interestingly, the semantics of $\mathcal{L}$FSQL can also be used to investigate functional dependencies more efficiently.

1.2 Main Contributions of the Thesis

In this section, we will state the main contributions of this work. We will follow it by emphasizing on how this thesis is organized.

One of the main contributions of this thesis is that we provide a formal definition of the syntax of $\mathcal{L}$FSQL. We also give concrete semantics of $\mathcal{L}$FSQL in $\mathcal{L}$-fuzzy relations. Last but not the least, we provide an implementation of a parser for $\mathcal{L}$FSQL in Haskell. This thesis is related to the work of Chowdhury [2].

This thesis is organized in six chapters. Chapter 1 provides general introductory information about this work. In Chapter 2, we provide background information about relational databases, SQL, Fuzzy databases and FSQL. We further introduce $\mathcal{L}$-fuzzy relations and $\mathcal{L}$FSQL. Chapter 3 provides mathematical concepts behind $\mathcal{L}$-fuzzy databases and $\mathcal{L}$FSQL. Chapter 4 mainly discusses the definitions of $\mathcal{L}$-fuzzy database and $\mathcal{L}$FSQL. It provides explanations about the various elements of $\mathcal{L}$-fuzzy databases and $\mathcal{L}$FSQL, the grammar guiding the creation of the $\mathcal{L}$FSQL, and the semantics of $\mathcal{L}$-fuzzy database and $\mathcal{L}$FSQL. The actual implementations of $\mathcal{L}$-fuzzy database and $\mathcal{L}$FSQL in terms of datatypes and the main functions are discussed in Chapter 5. Lastly, conclusions are given in Chapter 6.

Chapter 2

Preliminaries

In this chapter, we give a short discussion of regular relational databases and fuzzy relational databases to aid the reader's understanding of this work. We briefly introduce the actual work which is the $\mathcal{L}$-fuzzy relational database and its standard language $\mathcal{L}$FSQL.

2.1 Relational Database and SQL

In this section, we take a look at regular relational databases developed by Codd 1975 as well as the language SQL developed by Chamberlin and Boyce in the 1970s. We consider how the regular relational databases and SQL are viewed in the mathematical sense. We give an example of regular or crisp relations. We represent the relations in matrix form to show how crisp relations are treated through Boolean filters. In addition, we give some examples of SQL statements.

2.1.1 Relational Database

The relations in a relational model for databases are similar to regular relations used in mathematics [3]. A relational database (RDB) consists of a set of relations (tables) with sets (columns or attributes) and unique instances (rows or records) with values for each attribute. A table in an RDB can be viewed as a cross product of the attributes (sets). In a mathematical sense, a classical relation is the set of ordered pairs. Notationally, if S and T are sets, then relation R between S and T is a subset of the Cartesian product $S \times T = \{(a, b) \mid a \in S, b \in T\}$, which is $R \subseteq S \times T$. A relationship can either exist or does not exist between elements [25, 26]. A relation can

also be represented by its characteristic function $\mu_R : S \times T \to \mathbb{B}$ where $\mathbb{B}$ represents the Boolean values [14]. The characteristic function is defined by$\mu_R(a, b)$= true iff $(a, b) \in R$.

Example 2.1.1. Let us consider Table 2.1 as database table which represents students' records concerning their interest in some courses. Student Id (Std_Id) = {S1, S4, S3, S2, S5}. On the other hand, Interest = {Political Science (P.Sci), Computing (Comp), Geography (Geog), Mathematics (Maths)}. The students' records relation or table can be viewed in the mathematical sense as a subset of Std_Id × Interest i.e. Students = {(S1,P.Sci), (S4,Comp), (S3,Geog), (S2,Maths), (S5,Comp)}

Std_Id	**Interest**
S1	P.Sci
S4	Comp
S3	Geog
S2	Maths
S5	Comp

Table 2.1: Students' records

Matrix Representation

The representation of the relation students' records in the matrix form from Figure 2.1 gives a clear pictorial view of how crisp relations are represented through Boolean values. The Boolean values demonstrate if there is a relation between the two elements or not. We use 1 and 0 for true and false respectively. The pair (S2, Maths) belongs to the relation Students and it will have a Boolean value of 1. The pair (S4, P.Sci) which does not exist in the relation Students will have a Boolean value of 0.

$$\text{Students} = \begin{array}{c} \\ S1 \\ S4 \\ S3 \\ S2 \\ S5 \end{array} \begin{array}{c} \begin{array}{cccc} \textit{P.Sci} & \textit{Comp} & \textit{Geog} & \textit{Maths} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2.1: Matrix representation of students records

Table 2.1 is a binary relation as it has exactly two columns. Relational database

tables normally contain more than 2 columns. To represent such a crisp relation in a Boolean matrix, is a matter of having a break point in the pairs by grouping some pairs as the source and the remaining as the target.

The relation $A \times B \times C$ represented by Table 2.2 can be viewed in a different form as $n \to A \times B \times C$ where n is the number of rows in the table. Figure 2.2 shows the representation of Table 2.2 in the Boolean matrix. It gives a pictorial view of how such a table with more than 2 columns is represented in the database. The source {1,2,3,4} is coming from n and the target {xxx, xxy, xyx, xyy, yxx, yxy, yyx, yyy} is a subset of the power set of $A \times B \times C$. Each row represents an object or record in the database table.

A	***B***	***C***
x	x	y
x	y	y
y	x	x
y	y	y

Table 2.2: Multiple column table

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccccc} xxx & xxy & xyx & xyy & yxx & yxy & yyx & yyy \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Figure 2.2: Matrix representation of the relation $n \to A \times B \times C$.

2.1.2 Structured Query Language

We can view SQL as the standard language used to interact with the RDBMS which contains, controls and manages the database's resources and data. SQL is one of the reasons for the popularity of the RDB due to its English like structure and the intuitive meaning of its operations [5]. The foundations of SQL were based on the theories in the relational model. Some of the operators upon which SQL was built, were already existing in mathematics before the relational model for databases was invented [3]. Some of these mathematical operators in relational model theories upon which SQL was built are: union, intersection, division and Cartesian product. An SQL **INSERT** statement in the mathematical sense is the union of the data in the

selected table from the database and row (record) formed by the values in the insert statement.

Example 2.1.2. We now perform some examples of SQL queries. The tables below show the original table and the resulting tables from SQL **SELECT**, **DELETE** and **INSERT** queries. Table 2.3 is the actual students table on which we carried out the SQL queries.

Table 2.4 is the resulting table from inserting the record "S6, Maths" by the **INSERT** statement

INSERT INTO Students **VALUES**(S6, Maths);

Similarly, Table 2.5 is the resulting table from deleting record "S3, Geog" from the students table by the statement

DELETE FROM Students **WHERE** Std_Id = S3;

Lastly, Table 2.6 represents the result of selecting some records from the Students table by

SELECT Std_Id, Interest **FROM** Students
WHERE Std_Id = S4 **OR** Std_Id = S5;

Std_Id	**Interest**
S1	P.Sci
S4	Comp
S3	Geog
S2	Maths
S5	Comp

Table 2.3: Students table

Std_Id	**Interest**
S1	P.Sci
S4	Comp
S3	Geog
S2	Maths
S5	Comp
S6	Maths

Table 2.4: Result table from the **INSERT** statement.

Std_Id	**Interest**
S1	P.Sci
S4	Comp
S2	Maths
S5	Comp

Table 2.5: Result table from the **DELETE** statement.

Std_Id	**Interest**
S4	Comp
S5	Comp

Table 2.6: Result table from the **SELECT** statement.

Both relational databases and SQL are based on the crisp concept. The crisp concept uses an all inclusive or all exclusive approach to deal with membership of elements. They are incapable of representing, handling and manipulating imprecise or ambiguous data well. There have been several attempts by researchers in the relational model to deal with the representation and manipulation of fuzzy data in RDB and its SQL. Each approach has its advantages and disadvantages. Codd introduced the NULL value as a value to be used in place of imprecise and missing data [6]. A comparison of any value with the null value yields neither true or false, but rather yields maybe as the result [6].

2.2 Fuzzy Relational Database and FSQL

This section contains the discussion of the basic concepts underlying fuzzy relational databases. We take a view on how fuzzy relational databases treat fuzzy data. Furthermore, we provide a matrix representation of a relation using matrices with coefficients from the unit interval [0,1]. Lastly, we discuss FSQL the standard language for fuzzy relational databases. We take a look at some of the elements which have been added to SQL to form FSQL in order to handle fuzzy data.

2.2.1 Fuzzy Relational Database

FSQL and FRDB are both based either "on fuzziness" or "on the fuzzy concept" introduced by Zadeh in 1965 [6, 32]. Fuzzy sets or relations go beyond the crisp concept of either all-inclusive or all-exclusive by mapping elements to membership degrees taken from the interval [0,1]. A fuzzy set A can be denoted mathematically as $A=\{\mu_A(x_i)/x_i, \ldots, \mu_A(x_n)/x_n\}$ where $\mu_A(x_i) \in [0,1]$ is the membership degree and $x_i \in X$ where X is the underlying universe of discourse. An element which belongs to a fuzzy set with degree 1 is a full member whilst an element with membership degree 0 is absolutely not a member of the fuzzy set. A membership degree between 0 and 1 determines the level of membership.

The fuzzy concept has paved the way for the use of imprecise and ambiguous elements as values in addition to crisp values in databases. Linguistic labels can be defined on any attribute which can be treated as fuzzy set [6, 8, 11]. These labels are already defined in the fuzzy meta-knowledge base. For example, Figure 2.3 shows the definition

of linguistic labels **$Young, $Old** and **$Very_Old** on Age. The x-axis represents the actual age whilst the y-axis represents the membership value of the actual age in the linguistic labels. A person with age 30 will belong to **$Young** or **$Old** with membership degree 0.5 but will belong to **$Very Old** with membership degree 0.

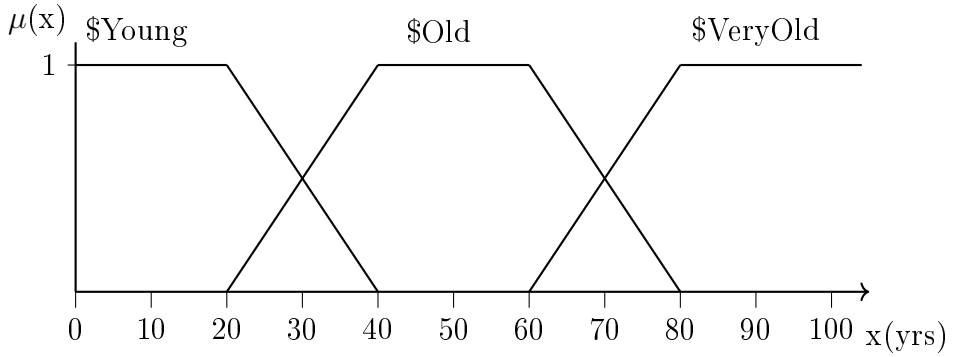


Figure 2.3: Age distribution in terms of linguistic labels

Similar to the classical relation representation through a characteristic function, a fuzzy relation can be seen as a map from X to a real number between 0 and 1 inclusively indicating the degree of its existence. In other words, a relation with membership degree 1 indicates that there is a strong relation between the elements whilst the relation with degree 0 indicates that there is no relation. Similarly, a relation with membership degree in between 0 and 1 represents the level of relationship between the elements. Crisp relations are a special case of fuzzy relations. They are fuzzy relations where the membership values are restricted to $\{0,1\}$. In mathematical representation, a fuzzy relation R between the sets A and B is a function $R : A \times B \rightarrow [0, 1]$.

Matrix Representation of Fuzzy Relations

Example 2.2.1. Consider the Example 2.1.2, where interest can be assumed as fuzzy in nature. Interest can be visualized as a qualitative attribute and is subjective as well. There can be several levels of interest in a course. Students may have more interest, little interest in a course. In a crisp model, either you have absolute interest to have Boolean value 1 or risk of having Boolean value 0 for any other level of interest. It is best to represent such a relation as a fuzzy relation. Figure 2.4 shows the matrix representation to illustrate a fuzzy relation on the students' records.

$$\text{Students} = \begin{array}{c} \\ S1 \\ S4 \\ S3 \\ S2 \\ S5 \end{array} \begin{array}{c} \begin{array}{cccc} \textit{P.Sci} & \textit{Comp} & \textit{Geog} & \textit{Maths} \end{array} \\ \begin{pmatrix} 1 & 0.43 & 0.75 & 0.10 \\ 0 & 1 & 0.65 & 0.85 \\ 0.89 & 0.30 & 1 & 0 \\ 0 & 0.95 & 0.38 & 1 \\ 0.64 & 1 & 0 & 0.51 \end{pmatrix} \end{array}$$

Figure 2.4: Matrix representation of students' records in fuzzy relation

From Figure 2.4, some of the students who could not give affirmative responses of their interest in some of the courses are well represented. In other words, the fuzzy form gives the chance to represent the different levels of interest with respect to the students' responses. Those responses such as "I have little interest", "maybe I have interest" and "I have more interest" will get degrees between 0 and 1 inclusively instead of 1 for "I have interest" and 0 for any other level of interest. A fuzzy table with more than two columns is treated the same way as the classical table as being explained in Section 2.1.1. The only difference is, instead of Boolean values 0 or 1 as membership degrees, the unit interval [0,1] is used.

2.2.2 FSQL

Fuzzy Structured Query language is used to query a fuzzy database. It is an extension of SQL and all the valid statements in SQL are also valid in FSQL [6]. In order words, to use FSQL to represent and manipulate imprecise data, additional elements have been added to the SQL to form FSQL. The **SELECT** statement for instance, which is one of the complex statements in the SQL has some additional elements such as fulfillment threshold (**Thold**), compatibility degree (**CDEG()**) and Fuzzy comparators. A **SELECT** query in FSQL could be:

SELECT * FROM Students, **CDEG() WHERE** Age **FEQ** $Young **THOLD** 0.5

Fulfillment threshold and its value (**THOLD** 0.5) in FSQL indicates the minimum degree (0.5) that a tuple must satisfy a given condition to be part of the resultant. In fuzzy theory, it represents an α-cut which can be viewed as a fuzzy property which sets the membership degree of elements greater or equal to the α-cut value to degree 1 and the rest to degree 0. More information about α-cut will be discussed in the next chapter. Compatibility degree **CDEG()** is a function that computes the degree of satisfaction of tuples to the **SELECT** query condition.

Fuzzy Equal (**FEQ**) is an example of fuzzy comparators used to compare an attribute with attribute of the same type or attribute and a value. In the **SELECT** query above, (**FEQ**) is comparing attribute Age and the fuzzy value **$Young**. There are 18 comparators in addition to the crisp comparators. Eight of them are possibility comparators which compare elements based on their general features. The next eight are necessity comparators, which are the corresponding version of the possibility comparators. The necessity comparators compare elements based on specific features only. The possibly equal comparator returns true if at least some of the compared features are equal. In the case of necessarily equal, it returns true only if all the comparing features are the same. In effect, the possibly equal selects more tuples than necessarily equal. The remaining two are inclusion and fuzzy inclusion.

There are also logical operators **Or**, **And** and **Not** in FSQL. The t-norm and the t-conorm are functions that provide a lot of operators in fuzzy concept [6]. In the fuzzy concept, the **Or** operator computes by using a t-conorms or s-norms and by default, the maximum is used. On the other hand, the **And** operator computes by using a t-norms and by default the minimum is used. The **Not** operator also operates by using negation or the complement of fuzzy set [6]. There are other functions in t-norm and t-conorm that the user of FSQL can specify instead of using the default ones [6]. Fuzzy databases and FSQL both make it possible to handle and manipulate inexact data. They give the flexibility to handle both quantitative and qualitative information. Fuzzy databases and FSQL can only handle linear order data well but cannot handle partial ordered data as being explained in the introduction of this thesis.

Example 2.2.2. Let us consider an example of FSQL statements on members' records of a club as shown in the tables below. Figure 2.5 is a trapezoidal diagram showing the Age distribution in terms of fuzziness whilst Table 2.7 is the members' records table. We assume that from the Age distribution, 30 years and 40 years old persons have a degree of membership 0.5 and 0 respectively in the linguistic label **$Young**. We also assume that 40 years has degree 1 in the label **$Old**.

Table 2.8 is the resulting table from inserting a new member record with M5 and **$Old** fuzzy values as Id and Age respectively. We used the **INSERT** statement:

INSERT INTO Members **VALUES**(M5, **$Old**);

We also deleted records "M3, 30" and "M4, 40" from Table 2.7. The Table 2.9 is

the resulting table from the **DELETE** query. Although, age 30 is indeed part of **$Young** but the degree is less than the threshold. We achieved Table 2.9 by the **DELETE** statement:

DELETE FROM members **WHERE** Age **NFLEQ $Young Thold** 0.8;.

Similarly, Table 2.10 is the resulting table from the **SELECT** statement:

SELECT Std_Id, Age **CDEG FROM** Members **WHERE** Age **FEQ $Young Thold** 0.4;

Using the condition in the **SELECT** statement above to consult Figure 2.5, age 30 yrs, 40 yrs and **$Young** have degree 0.5, 0 and 1 respectively in the set **$Young**. The additional restriction from the threshold which is 0.4 discarded the record with Age 40 yrs. The record with Age 40 yrs has a degree less than the threshold.

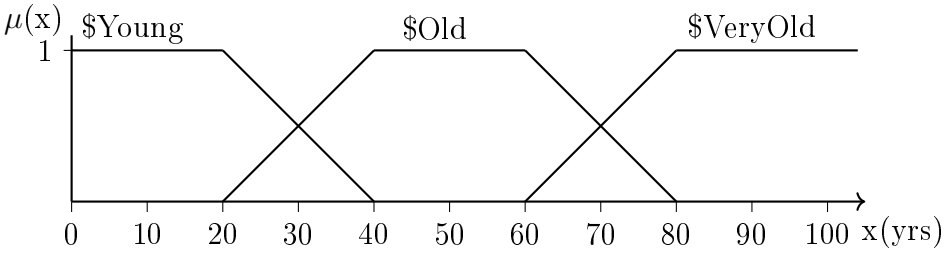


Figure 2.5: Trapezoidal diagram of Age distribution

Mem_Id	Age
M1	$Young
M4	40
M3	30

Table 2.7: Club members table

Mem_Id	Age
M1	$Young
M4	40
M3	30
M5	$Old

Table 2.8: Result table from the FSQL **INSERT** statement.

Mem_Id	Age
M1	$Young

Table 2.9: Result table from the FSQL **DELETE** statement.

CDEG	Mem_Id	Age
1	M1	$Young
0.5	M4	30

Table 2.10: Result table from the FSQL **SELECT** statement.

2.3 $\mathcal{L}$-Fuzzy Relations and $\mathcal{L}$FSQL

In this section, we begin our discussion by taking a brief look at $\mathcal{L}$-fuzzy relations. We consider the underlying lattice theory and some of its operations. We give an $\mathcal{L}$-fuzzy relation version of the student's records and represent it in the matrix form as well. Lastly, we consider a brief look at $\mathcal{L}$FSQL and its nature with respect to the already existing languages SQL and FSQL.

2.3.1 $\mathcal{L}$-Fuzzy Relations

$\mathcal{L}$-fuzzy relations map elements to a value of an arbitrary lattice $\mathcal{L}$ instead of the unit interval [0,1]. Though the fuzzy concept is capable of handling imprecise data which are linear, sometimes fuzzy data can be nonlinear [9]. In comparing elements, some elements may be incomparable. In other words, none precedes the other. In such a situation the ideal representation will be using $\mathcal{L}$-fuzzy sets.

For example, let us consider a case where you have multiple factors to contemplate before you can optimize your choice and some of these factors are not related to each other or are incomparable. In order to optimize such a choice, the criteria should form a complete lattice at least. A complete lattice is partially ordered and all the subsets must have both meet ($\wedge$) and join ($\vee$) [9]. A partially ordered set is a set with an order relation, which is a relation that is reflexive, transitive and antisymmetric. Figure 2.6 is a Hasse diagram which is a mathematical structure representing a lattice partially ordered by inclusion on the set $A = \{a, b, c\}$. We give a more detailed discussion on posets in Chapter 3.

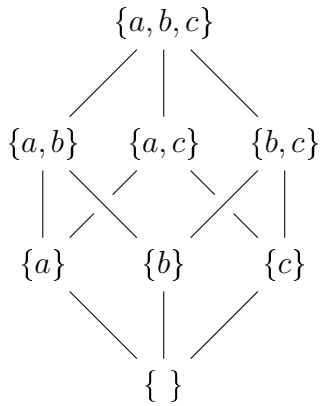


Figure 2.6: Lattice by inclusion on set $A = \{a, b, c\}$

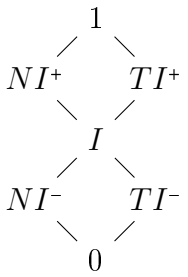
From Figure 2.6, there is a relation between $\{a\}$ and $\{a, b\}$, which is $\{a\}$ is included in $\{a, b\}$. On the hand, there is no relationship between $\{a\}$ and $\{b\}$ under the inclusive definition. In other words, you can not say that $\{a\}$ is included in $\{b\}$ or vice versa. However, for every subset of the lattice, there exists a meet and a join. For example, $\{a\} \vee \{b\} = \{a, b\}$ and $\{a\} \wedge \{b\} = \{\ \}$ or the empty set.

In general lattice theory, lower bounds or upper bounds or both may exist for a subset of a given ordered set. The least among the upper bounds is called least upper bound or supremum, and the greatest among the lower bounds is the greatest lower bound or infimum of the subset. The greatest element in the whole set is the least upper bound of the set. Likewise, the least element of the set is the greatest lower bound of the set. In Figure 2.6, the least element and the greatest element of the lattice are $\{\}$ and $\{a, b, c\}$ respectively. Given any element x, the meet of x and the greatest element always results in x whilst the join results in the greatest element. On the contrary, the meet of x and the least element always results in the least element but the join results in x. The laws and properties of lattices can be extended to $\mathcal{L}$-fuzzy relations [9]. A more mathematically detailed explanation of the concepts about posets as well as lattices are given in the next chapter.

$\mathcal{L}$-fuzzy relations can be viewed also as a characteristic function. Mathematically, an $\mathcal{L}$-fuzzy relation between A and B is a function $A \times B \rightarrow \mathcal{L}$ where $\mathcal{L}$ denotes the lattice of membership values. $\mathcal{L}$-fuzziness is a generalization of fuzziness which includes both crispness and the original fuzziness by Zadeh. If $\mathcal{L} \equiv \{0,1\}$, we are dealing with crispness [4]. This can be referred as Boolean lattice of truth-values [25]. Again, if $\mathcal{L} \equiv [0,1]$, then we are dealing with the original fuzziness introduced by Zadeh [4].

2.3.2 Matrix Representation of $\mathcal{L}$-Fuzzy Relations

Let us consider again the students' records illustration. What if we want to know whether the student has taken the course before or not in connection with their interest in the course? Figure 2.7.(a) represents the given lattice $\mathcal{L}$ which is the set of responses from the students. The members of $\mathcal{L}$ can be viewed in more detail as absolutely not interested (0), taken but mostly uninterested (TI^-), not taken but mostly not interested (NI^-), Interested (I), not taken but mostly interested (NI^+), taken but mostly interested (TI^+) and absolutely interested (1). Figure 2.7.(b) shows the matrix representation of the students' records illustration in $\mathcal{L}$-fuzzy relation.



(a) Lattice distribution of students' reponses

	P.Sci	*Comp*	*Geog*	*Maths*
$S1$	1	TI^+	NI^+	TI^-
$S4$	0	1	I	TI^+
$S3$	NI^+	TI^-	1	0
$S2$	0	NI^+	NI^-	1
$S5$	TI^+	1	0	I

(b) Matrix representation of students' records in L-fuzzy relation

Figure 2.7: $\mathcal{L}$-fuzzy version of students' records

With respect to Figure 2.7, it is clear that we can combine several factors in an $\mathcal{L}$-fuzzy approach that could not be modeled in the fuzzy approach by Zadeh. We have been able to add whether the student has taken the course or not as additional factor in determination of the students interest in the course. The lattice can be restructured or extended to suite our needs for membership values without worrying about the unordered structure some factors may pose. The $\mathcal{L}$-fuzzy approach also gives a richer notation than crisp and fuzzy approaches [14]. For example, instead of using only 0s and 1s or the unit interval, from Figure 2.7, we have been able to use TI^+, I and NI^- without any difficulties or complexities. With regard to matrix representation of tables with more than two columns, we treat it the same as in Figure 2.2. The only difference is instead of 0s and 1s, we use arbitrary lattice values.

2.3.3 $\mathcal{L}$FSQL

$\mathcal{L}$FSQL will be the query language to be used to represent and manipulate $\mathcal{L}$-fuzzy data in the fuzzy relational database. It will be the extension of the FSQL to accommodate non linear fuzzy data. Instead of membership values from [0,1] only, $\mathcal{L}$FSQL will take values from an arbitrary lattice which also contains the crisp values as well as the fuzzy values. We want to present a small core language which is based on SQL and FSQL.

SQL and FSQL were based on the mathematical operators and properties in the classical sets and fuzzy sets respectively. Likewise, $\mathcal{L}$FSQL will be based on the laws, properties and operators of $\mathcal{L}$-fuzzy sets such as meet, join, reflexivity, transitivity and asymmetry to execute its queries statements. The next chapter will give more detail

of the underlying operators and properties for $\mathcal{L}$FSQL. We will give some examples of $\mathcal{L}$FSQL and how they work in Chapter 4.

Chapter 3

Mathematical Preliminaries

In this chapter, we discuss the underlying mathematical framework for this work. We define the various concepts, properties and give concrete examples to explain these concepts and properties. We will use "$\vee$" and "$\wedge$" to denote join and meet respectively in lattices and $\mathcal{L}$-fuzzy sets.

3.1 Lattices

This section basically contains definitions, properties and some concrete examples of lattices. We start by defining lattices and looking at some of their properties and operators. We further the discussion by looking at distributive lattices and Heyting algebras.

3.1.1 Lattices

A rule (relation) such as "contains in ($\subseteq$)" or "less or equal ($\leq$)" is normally called an order relationship, or an order for short. The underlying set together with the order is known as a poset [1, 10]. Formally, we have the following definition.

Definition 3.1.1. A set A with a binary relation $\leq$ is a poset if and only if for all a, b, c the properties from 1 to 3 below are satisfied.

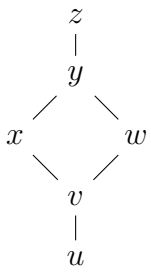
(1) $a \leq a$ Reflexive

(2) iff $a \leq b$ and $b \leq c$, then $a \leq c$ Transitive

(3) iff $a \leq b$ and $b \leq a$, then $a = b$ Antisymmetric

Example 3.1.2. Consider poset $A = \{u, v, w, x, y, z\}$ defined by $\leq$ relation as shown in Figure 3.1. From the matrix representation, the leading diagonal indicates that every element in A is $\leq$ itself, thus, satisfying reflexivity. Again, none of the elements are related in the reverse order unless the elements are equal which indicates the relation is antisymmetric. Lastly, from Figure 3.1.(b), $u \leq v$ and $v \leq y$ and so we can conclude that $u \leq y$ is transitive.

In the Hasse diagram Figure 3.1.(a), we only draw the necessary line segments. We did not show the line segments for transitivity and reflexivity but the bottom line is the elements are ordered. In other words, the element at the lower end of the line segment is less than the one at the upper end.



(a) Graphical representation of poset A under the relation $\leq$

$$\begin{array}{c} \leq \\ u \\ v \\ w \\ x \\ y \\ z \end{array} \begin{array}{c} \begin{array}{cccccc} u & v & w & x & y & z \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

(b) Matrix representation of poset A under the relation $\leq$

Figure 3.1: Poset A under the relation $\leq$

In a poset some elements maybe be incomparable, for example x, w in Figure 3.1 (a). In addition to the above properties, if all the elements are comparable, the poset is referred to as linearly ordered or totally ordered. In other words, for all a, b we have $a \leq b$ or $b \leq a$ [1, 10].

Supremum and Infimum

A subset of poset may have upper bounds and lower bounds. The upper bounds of a subset B consists of all the elements in the poset A which are greater or equal to the elements in B. The least among the upper bounds is called least upper bound or supremum (sup) of the subset. We denote the least upper bound of B by sup B if it exists.

Definition 3.1.3. Suppose $(A, \leq)$ is a poset and $B \subseteq A$. Then we call $x \in A$ an upper bound of B iff $y \leq x$ for all $y \in B$, the least upper bound or supremum of B is x iff x is an upper bound of B and $x \leq z$ for every upper bound z of B.

Lower bounds on the other hand, are all the elements in the poset A which are less or equal to the elements in B. Similarly, the greatest among the lower bounds is called greatest lower bound or infimum (inf). We denote the greatest lower bound of B by inf B if it exists.

Definition 3.1.4. Again let us assume $(A, \leq)$ is a poset and $B \subseteq A$. Then we call $x \in A$ a lower bound of B iff $y \leq x$ for all $y \in B$, the greatest lower bound or supremum of B is x iff x is a lower bound of B and $x \leq z$ for every lower bound z of B.

In Figure 3.1.(a), $\inf\{x, w\} = v$ and $\sup\{x, w\} = y$. Depending on the structure of the subset, infimum and supremum may exist or may not exist for the subset. There is always a greatest and a least element of poset and they form the least upper bound and the greatest lower bound respectively of the poset. From the Figure 3.1.(a), the greatest element or supremum of poset A is z and the least or infimum is u. In an empty set, the greatest element is the same as the least element.

Now, based on the above properties, we can define lattice as a partially ordered set where for every pair of elements in the set, there exist an infimum and a supremum.

Definition 3.1.5. A poset $\langle \mathcal{L}, \leq \rangle$ is a lattice iff $\mathcal{L}$ is a non-empty set, $\inf\{a, b\}$ and $\sup\{a, b\}$ exist for all $a, b \in \mathcal{L}$ [1, 10].

Lattices can also be defined algebraically in terms of the meet ($\wedge$) and join ($\vee$) operations.

Definition 3.1.6. A triple $\langle \mathcal{L}, \vee, \wedge \rangle$ is lattice iff meet ($\wedge$) and join ($\vee$) are binary operators on $\mathcal{L}$ and both $\vee$ and $\wedge$ satisfy the properties of commutativity, associativity, idempotency and the two absorption identities, [10]. For all x, y, z we have:

(4) $x \vee y = y \vee x$ and $x \wedge y = y \wedge x$ Commutative

(5) $x \vee (y \vee z) = (x \vee y) \vee z$ and

(6) $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ Associative

(7) $x \vee x = x$ and $x \wedge x = x$ Idempotency

(8) $x \wedge (x \vee y) = x$ and $x \vee (x \wedge y) = x$ Absorption.

Both definitions, the order based and the algebraic definitions for example, $x \wedge y = \inf\{x, y\}$ are equivalent as shown in [10]. Examples of lattices are Figure 2.5 at Section 2.3.1 and Figure 2.6.(a) at Section 1.3.2. However, Figure 3.2 is an example of a poset which is not a lattice. Although, there are upper bounds $\{d, e, f\}$ for b and c but the least upper bound does not exist. In the same manner, there are lower bounds $\{a, b, c\}$ for d and e but the greatest lower bound does not exist.

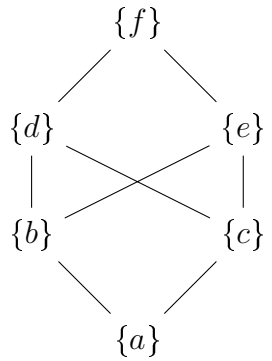


Figure 3.2: An example of a non-Lattice poset

Definition 3.1.7. A lattice with both an infimum and a supremum for every subset is called a complete lattice.

From the Definition 3.1.7, we can now define a bounded lattice which will help us to include crispness as well as restricting the boundaries of the lattice. Algebraically, we can define bounded lattice as:

Definition 3.1.8. A lattice $\langle \mathcal{L}, \vee, \wedge \rangle$ is a bounded lattice iff 0 and $1 \in \mathcal{L}$ and satisfy the following properties: for all $a \in \mathcal{L}$,

(9) $a \wedge 1 = a$ identity on $\wedge$

(10) $a \wedge 0 = 0$ dominancy on $\wedge$

(11) $a \vee 1 = 1$ dominancy on $\vee$

(12) $a \vee 0 = a$ identity on $\vee$

Every complete lattice is bounded.

3.1.2 Distributive lattice

From the Definition 3.1.6, we can be tempted analogically to assume that meet and join binary operators have the same distributive property in lattice as the arithmetic binary operators multiplication (*) and addition (+) have, but unfortunately not all lattices preserve the distributive properties of meet and join. Thus, any lattice that includes M_3 or N_5 shown in Figure 3.3 as a sublattice does not satisfy the distributive property, hence is not distributive lattice [10, 21].

(a) M_3 a non-distributive lattice (b) N_5 a non-distributive lattice

Figure 3.3: Diagrammatical view of M_3 and N_5.

In Figure 3.3 (a), $b \wedge (d \vee c) = b \wedge e = b$ but $(b \wedge d) \vee (b \wedge c) = a \vee a = a$ i.e., $b \wedge (d \vee c) \neq (b \wedge d) \vee (b \wedge c)$. Similarly, from Figure 3.3 (b), $d \wedge (b \vee c) = d \wedge e = d$ but $(d \wedge b) \vee (d \wedge c) = b \vee a = b$ i.e., $d \wedge (b \vee c) \neq (d \wedge b) \vee (d \wedge c)$

Definition 3.1.9. A lattice is distributive iff for all x, y, $z \in \mathcal{L}$,

(13) $x \wedge (y \vee z) = (x \vee y) \wedge (x \vee z)$ $\wedge$ distributing over $\vee$

(14) $x \vee (y \wedge z) = (x \wedge y) \vee (x \wedge z)$ $\vee$ distributing over $\wedge$

The above properties in the Definition 3.1.9 dually imply each other. That is iff Property (13) holds for all elements of $\mathcal{L}$, it implies Property (14) will also hold for all elements of $\mathcal{L}$ and vice versa [1].

3.1.3 Heyting Algebras

A Heyting algebra, also called a relative pseudo-complemented lattice, is a bounded lattice equipped with a binary implication operator ($\rightarrow$).

Definition 3.1.10. A bounded lattice $B = (\mathcal{L}, \vee, \wedge)$ is Heyting algebra iff there is an implication operator $(\rightarrow)$ such that for all $a, b, c \in \mathcal{L}$
$a \rightarrow a = 1$
$a \wedge (a \rightarrow b) = a \wedge b$
$b \wedge (a \rightarrow b) = b$
$a \rightarrow (b \wedge c) = (a \rightarrow b) \wedge (a \rightarrow c)$

The relative pseudo-complement $a \rightarrow b$ of a and b can also be characterized by $x \wedge a \leq b$ iff $x \leq a \rightarrow b$. In other words, it is the greatest element x such that $x \wedge a \leq b$.

Example 3.1.11. Let us compute the relative pseudo-complement of some of the elements in the lattice structure shown in Figure 3.4. To compute $\alpha \rightarrow \beta$ for example, we take the meet of α and all the various elements in the lattice. We then select all the elements whose resulting value from the meet with α is less than β.
First we obtain:

$\alpha \wedge 0 = 0$
$\alpha \wedge \alpha = \alpha$
$\alpha \wedge \beta = 0$
$\alpha \wedge \gamma = \alpha$
$\alpha \wedge \delta = 0$
$\alpha \wedge 1 = \alpha$

Now, we compute the supremum of all elements x so that $\alpha \wedge x \leq \beta$. We get $0 \vee \beta \vee \delta = \delta$ so that $\alpha \rightarrow \beta = \delta$.

Following the same procedure used to compute $\alpha \rightarrow \beta$, we can obtain the relative pseudo-complement of $\alpha \rightarrow \alpha$ and $\beta \rightarrow \alpha$ as 1 and α respectively.

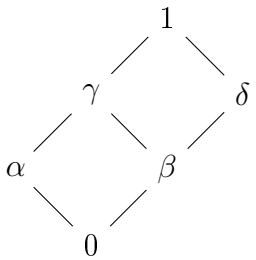


Figure 3.4: Lattice structure

3.2 $\mathcal{L}$-Fuzzy Sets

After focusing our discussion on lattices in the above subsection, we now proceed with our discussion by looking into $\mathcal{L}$-fuzzy sets. We define $\mathcal{L}$-fuzzy sets and their operators meet and join.

3.2.1 $\mathcal{L}$-Fuzzy Sets

From now on, $\mathcal{L}$ is always a complete Heyting algebra. Sometimes, due to the structure of elements in a set, we have to map the elements to arbitrary lattice in order to obtain an optimum value. To evaluate such elements in the set, the lattice must be a complete lattice [9]. An $\mathcal{L}$-fuzzy set is a set with arbitrary lattice membership values.

Definition 3.2.1. An $\mathcal{L}$-fuzzy subset A of X is function $A : X \to \mathcal{L}$

From the Definition 3.2.1, we can also view $\mathcal{L}$-fuzzy sets algebraically as all functions from $X \to \mathcal{L}$, which is $\mathcal{L}^X$ where $\mathcal{L}$ is the lattice and X is the underlying universe of discourse. The algebraic structure of $\mathcal{L}$ is the same as $\mathcal{L}^X$ and if and only if the functions of $\mathcal{L}$-fuzzy sets are equal, we can conclude that the sets are also equal [9]. The graph of a function f, which is $\text{graph}(f) = \{(a, f(a)) | a \in A\}$

Example 3.2.2. Let us consider the set $X = \{u, v, w, x, y, z\}$ and Figure 3.4 where $\mathcal{L}$ = {0, α, β, γ, δ, 1}. We can form the graph of $\mathcal{L}$-fuzzy sets:

$$\text{graph}(A) = \{(u,\gamma), (v,1), (w,\beta), (x,\delta), (y,0), (z,\alpha)\}$$
$$\text{graph}(B) = \{(u,\alpha), (v,\gamma), (w,0), (x,\delta), (y,\beta), (z,\beta)\}$$
$$\text{graph}(C) = \{(u,0), (v,\delta), (w,0), (x,1), (y,\beta), (z,\delta)\}.$$

We are pairing the elements in X with their lattice membership values.

3.2.2 Meet and Join

The properties, laws and operators of lattices are also valid for $\mathcal{L}$-fuzzy sets. We can define meet and join operations and also state their properties in $\mathcal{L}$-fuzzy subsets from a lattice as well. If 0 and U are the empty and universal subset respectively of the $\mathcal{L}$-fuzzy, then U is the universal identity and 0 is the universal dominant under the meet operation. On the contrary to meet, 0 and U are the universal identity and

dominant respectively in join.

Definition 3.2.3. Suppose A and B are $\mathcal{L}$-fuzzy subsets of X. Then we define the meet and join of A and B, the universal set U and the empty set 0 for all $x \in X$ by:

$(A \vee B)(x) = A(x) \vee B(x)$
$(A \wedge B)(x) = A(x) \wedge B(x)$.
$U(x) = 1$
$0(x) = 0$

We will list some basic properties in the Lemma below.

Lemma 3.2.4. For a given $\mathcal{L}$-fuzzy sets A, B and C, we have

1. $A \vee B = B \vee A$ and $A \wedge B = B \wedge A$ **(4)**

2. $(A \vee B) \vee C = A \vee (B \vee C)$ **(5)**

3. $(A \wedge B) \wedge C = A \wedge (B \wedge C)$ **(6)**

4. $A \vee A = A$ and $A \wedge A = A$ **(7)**

5. $A \wedge U = A$ **(9)**

6. $A \wedge 0 = 0$ **(10)**

7. $A \vee U = U$ **(11)**

8. $A \vee 0 = A$ **(12)**

Below are concrete examples of meet and join based on the Example 3.2.2.

$\text{graph}(A \wedge B) = \{(u,\alpha), (v,\gamma), (w,0), (x,\delta), (y,0), (z,0)\}$

$\text{graph}(A \vee B) = \{(u,\gamma), (v,1), (w,\beta), (x,\delta), (y,\beta), (z,\gamma)\}$

$\text{graph}(A \wedge A) = \{(u,\gamma), (v,1), (w,\beta), (x,\delta), (y,0), (z,\alpha)\} = \text{graph}(A)$

$\text{graph}(A \wedge U) = \{(u,\gamma), (v,1), (w,\beta), (x,\delta), (y,0), (z,\alpha)\} = \text{graph}(A)$

$\text{graph}(B \wedge 0) = = \{(u,\alpha), (v,0), (w,0), (x,0), (y,0), (z,0)\}$

$\text{graph}(C \vee U) = \{(u,1), (v,1), (w,1), (x,1), (y,1), (z,1)\}$

$\text{graph}(B \vee 0) = \{(u,0), (v,\delta), (w,0), (x,1), (y,\beta), (z,\delta)\} = \text{graph}(B)$

3.2.3 Relative Pseudo-Complement

If y is the complement of x in a bounded lattice $\mathcal{L}$ for $x, y \in \mathcal{L}$ then $x \wedge y = 0$ and $x \vee y = 1$. Since the complement of x may not be unique, we result to relative pseudo-complement [30]. If we have a lattice, then there is a notation of relative pseudo-complement from Section 3.1.3. Since $\mathcal{L}$-fuzzy sets also form a lattice, they also do have relative pseudo-complements. We can compute the relative pseudo-complement on $\mathcal{L}$-fuzzy sets S and R by $(R \to S)(x) = R(x) \to S(x)$.

Example 3.2.5. Let us consider the graph of $\mathcal{L}$-fuzzy sets A and B below. Then we compute $(A \to B)(z)$ as $(A \to B)(z) = A(z) \to (z) = \alpha \to \beta = \delta$ where the last equality was already computed in Example 3.1.11.
graph(A) = {(u,γ), (v,1), (w,β), (x,δ), (y,0), (z,α)}
graph(B) = {(u,α), (v,γ), (w,0), (x,δ), (y,β), (z,β)}

3.3 $\mathcal{L}$-Fuzzy Relations

In this section we will look at $\mathcal{L}$-fuzzy relations. We will cover the transposition, composition and residual which are specifically related to $\mathcal{L}$-fuzzy relations. We will also consider "$\sqcup$" and "$\sqcap$" to represent join and meet respectively for $\mathcal{L}$-fuzzy relations. In the case of composition, we will use ";" and we read from left to right, eg. $R;Q$ will be read R followed by Q. We also provide a concrete example for easy understanding of these operations.

3.3.1 $\mathcal{L}$-Fuzzy Relations

Similarly to classical relations, we can view $\mathcal{L}$-fuzzy relations as characteristic functions that map pairs of elements from the cross product of sets to $\mathcal{L}$. The lattice $\mathcal{L}$ is used to determine the level of relation between the pairs. We can also view it algebraically as all functions from the pair A and B to $\mathcal{L}$ which is $\mathcal{L}^{A \times B}$. In other words, an $\mathcal{L}$-fuzzy relation is just an $\mathcal{L}$-fuzzy set of pairs.

Definition 3.3.1. An $\mathcal{L}$-fuzzy relation R is a function $R{:}A \times B \to \mathcal{L}$.

$\mathcal{L}$-fuzzy relations are $\mathcal{L}$-fuzzy sets and we can define the set-theoretic operators and

properties on relations as well [25]. From Example 3.3.2, we define some of the properties, operators and give concrete examples of $\mathcal{L}$-fuzzy relations.

Example 3.3.2. We will use the students' records illustration in $\mathcal{L}$-fuzzy relations from Section 2.3.2 to illustrate some of these properties and operations.

Mathematically, we can define $\mathcal{L}$ to be equal to {absolutely not interested (0), taken but mostly uninterested (TI^-), not taken but mostly not interested (NI^-), Interested(I), not taken but mostly interested (NI^+), taken but mostly interested (TI^+) and absolutely interested (1)}.

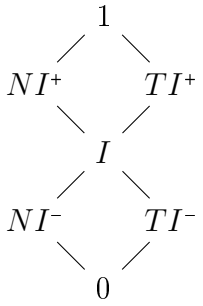


Figure 3.5: Lattice structure of students' responses

Let R represents the original $\mathcal{L}$-fuzzy relation from Section 2.3.2 and Q be a new $\mathcal{L}$-fuzzy relation on the same lattice above.

$$R = \begin{array}{c} \\ S1 \\ S4 \\ S3 \\ S2 \\ S5 \end{array} \begin{array}{c} \begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\ \begin{pmatrix} 1 & TI^+ & NI^+ & TI^- \\ 0 & 1 & I & TI^+ \\ NI^+ & TI^- & 1 & 0 \\ 0 & NI^+ & NI^- & 1 \\ TI^+ & 1 & 0 & I \end{pmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ S1 \\ S4 \\ S3 \\ S2 \\ S5 \end{array} \begin{array}{c} \begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\ \begin{pmatrix} 1 & I & TI^+ & NI^+ \\ NI^- & 1 & TI^- & NI^+ \\ NI^+ & I & 1 & 0 \\ 0 & TI^+ & NI^- & TI^- \\ TI^+ & TI^- & 0 & I \end{pmatrix} \end{array}$$

3.3.2 Meet and Join of $\mathcal{L}$-fuzzy Relations

We can take the meet or join of $\mathcal{L}$-fuzzy relations having their membership values from the same $\mathcal{L}$. $\mathcal{L}$-fuzzy relations are a special case of $\mathcal{L}$-fuzzy sets defined in Section 3.2. They are $\mathcal{L}$-fuzzy sets of pairs instead of $\mathcal{L}$-fuzzy sets of arbitrary values. We have already defined meet and join of $\mathcal{L}$-fuzzy sets. Here we give concrete example of meet and join in $\mathcal{L}$-fuzzy relations.

$(R \sqcap Q)(S4, P.Sci) = R(S4, P.Sci) \wedge Q(S4, P.Sci) = 0 \wedge NI^- = 0$
$(R \sqcap Q)(S1, Geog) = R(S1, Geog) \wedge Q(S1, Geog) = NI^+ \wedge I = I.$

$$R \sqcap Q = \begin{pmatrix} 1 & TI^+ & NI^+ & TI^- \\ 0 & 1 & I & TI^+ \\ NI^+ & TI^- & 1 & 0 \\ 0 & NI^+ & NI^- & 1 \\ TI^+ & 1 & 0 & I \end{pmatrix} \sqcap \begin{pmatrix} 1 & I & TI^+ & NI^+ \\ NI^- & 1 & TI^- & NI^+ \\ NI^+ & I & 1 & 0 \\ 0 & TI^+ & NI^- & TI^- \\ TI^+ & TI^- & 0 & I \end{pmatrix}$$

$$= \begin{array}{c} \\ S1 \\ S4 \\ S3 \\ S2 \\ S5 \end{array} \begin{array}{c} \begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\ \begin{pmatrix} 1 & I & I & TI^- \\ 0 & 1 & TI^- & I \\ NI^+ & TI^- & 1 & 0 \\ 0 & I & NI^- & TI^- \\ TI^+ & TI^- & 0 & I \end{pmatrix} \end{array}$$

With respect to join of $\mathcal{L}$-fuzzy relations, it is simply the join of the two degrees. From the $R \sqcup Q$ example,

$(R \sqcup Q)(S4, P.Sci) = R(S4, P.Sci) \vee Q(S4, P.Sci) = 0 \vee NI^- = NI^-$
$(R \sqcup Q)(S1, Geog) = R(S1, Geog) \vee Q(S1, Geog) = NI^+ \vee TI^+ = 1.$

$$R \sqcup Q = \begin{pmatrix} 1 & TI^+ & NI^+ & TI^- \\ 0 & 1 & I & TI^+ \\ NI^+ & TI^- & 1 & 0 \\ 0 & NI^+ & NI^- & 1 \\ TI^+ & 1 & 0 & I \end{pmatrix} \sqcup \begin{pmatrix} 1 & I & TI^+ & NI^+ \\ NI^- & 1 & TI^- & NI^+ \\ NI^+ & I & 1 & 0 \\ 0 & TI^+ & NI^- & TI^- \\ TI^+ & TI^- & 0 & I \end{pmatrix}$$

$$
\begin{array}{c} \\ S1 \\ S4 \\ = S3 \\ S2 \\ S5 \end{array}
\begin{array}{c}
\begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\
\left(\begin{array}{cccc}
1 & TI^{+} & 1 & NI^{+} \\
NI^{-} & 1 & I & 1 \\
NI^{+} & I & 1 & 0 \\
0 & 1 & NI^{-} & 1 \\
TI^{+} & 1 & 0 & I
\end{array} \right)
\end{array}
$$

3.3.3 Transposition, Composition and Residual

In this work, transposition, composition and residual are very important operations in the semantics specification of some of the component of $\mathcal{L}$FSQL such as the comparison operators.

Converse or transposition of $\mathcal{L}$-fuzzy relations are treated the same way we treat the converse of classical relations. We exchange the pair of elements that form the relation. In matrix representation, we swap the source and the target of the relation.

Definition 3.3.3. If $R : A \times B \to \mathcal{L}$ is an $\mathcal{L}$-fuzzy relation from A to B, then $R^T : B \times A \to \mathcal{L}$ is a relation from B to A and defined by $A^T(y, x) = A(x, y)$.

We compute the transpose of the $\mathcal{L}$-fuzzy relation R below.

$$
R^T = \begin{pmatrix}
1 & TI^{+} & NI^{+} & TI^{-} \\
0 & 1 & I & TI^{+} \\
NI^{+} & TI^{-} & 1 & 0 \\
0 & NI^{+} & NI^{-} & 1 \\
TI^{+} & 1 & 0 & I
\end{pmatrix}^{T}
$$

$$
= \begin{array}{c}
\begin{array}{ccccc} S1 & S4 & S3 & S2 & S5 \end{array} \\
\begin{array}{c} P.Sci \\ Comp \\ Geog \\ Maths \end{array}
\left(\begin{array}{ccccc}
1 & 0 & NI^{+} & 0 & TI^{+} \\
TI^{+} & 1 & TI^{-} & NI^{+} & 1 \\
NI^{+} & I & 1 & NI^{-} & 0 \\
TI^{-} & TI^{+} & 0 & 1 & I
\end{array} \right)
\end{array}
$$

The composition for classical relations is the multiplication of the two relations in the matrix form. In order to take the composition of relations, the rows of the first

relation must be equal to the columns of the second relation. We treat composition of $\mathcal{L}$-fuzzy relations the same way we treat the classical relations.

Definition 3.3.4. Given $\mathcal{L}$-fuzzy relations $T : A \times B \to \mathcal{L}$ and $S : B \times C \to \mathcal{L}$ the composition of T and S i.e $(T;S)(x,z) = \bigsqcup_{y \in B} \big(T(x,y) \wedge S(y,z)\big)$, for $x \in A$ and $z \in C$.

The following properties also hold for composition and transposition of $\mathcal{L}$-fuzzy relations. Given $\mathcal{L}$-fuzzy relations $S : A \to B, P : A \to B, R : B \to C, T : B \to C$ and $U : C \to D$

(15) $(T;S)^T = T^T;S^T$ Commutativity of composition and transposition

(16) $S;(T;U) = (S;T);U$ Associativity of composition

(17) $(S^T)^T = \mathrm{S}$ Involution

(18) $P \subseteq S$ and $R \subseteq T = P;R \subseteq S;T$

We give an example of composition of relations below. We strip the source and the target of the relations for simplicity.

$$R;R^T = \begin{pmatrix} 1 & TI^+ & NI^+ & TI^- \\ 0 & 1 & I & TI^+ \\ NI^+ & TI^- & 1 & 0 \\ 0 & NI^+ & NI^- & 1 \\ TI^+ & 1 & 0 & I \end{pmatrix}; \begin{pmatrix} 1 & 0 & NI^+ & 0 & TI^+ \\ TI^+ & 1 & TI^- & NI^+ & 1 \\ NI^+ & I & 1 & NI^- & 0 \\ TI^- & TI^+ & 0 & 1 & I \end{pmatrix}$$

$$= \begin{pmatrix} 1 & TI^+ & NI^+ & I & TI^+ \\ TI^+ & 1 & I & TI^+ & 1 \\ NI^+ & I & 1 & TI^- & I \\ I & 1 & 0 & 1 & NI^+ \\ TI^+ & 1 & 0 & I & 1 \end{pmatrix}$$

In the computation of $R;R^T$, $(R;R^T)(S_1,S_2)$ is computed as follows:

$(R;R^T)(S_1,S_2) = (1 \wedge 0) \vee (TI^+ \wedge 1) \vee (NI^+ \wedge I) \vee (TI^- \wedge TI^+) = TI^+$

Definition 3.3.5. Given $\mathcal{L}$-fuzzy relations $R : A \to B$, $S : B \to C$, and $T : A \to C$ for all $a \in A, b \in B, c \in C$ the left and the right residuals are defined by

$$(R\backslash T)(a,b) = \bigsqcap_{a \in A} \big(R(a,c) \to T(a,b)\big) \text{ and } (T/S)(a,b) = \bigsqcap_{c \in C} \big(S(a,c) \to T(b,c)\big)$$

respectively. The residuals can also be characterized by following equivalences:

$$R;X \subseteq T \iff X \subseteq R\backslash T \text{ and } Y;S \subseteq T \iff Y \subseteq T/S$$

For more information on the residuals and the proofs of these properties, we refer the reader to [29].

3.3.4 Least, Greatest and Identity Relations

Because of the definition of the least, greatest and identity relations, similar properties such as Properties **(9)** to **(12)** will also hold for $\mathcal{L}$-fuzzy relation. If the $\mathcal{L}$ in the $\mathcal{L}$-fuzzy relation is a bounded lattice, then

Definition 3.3.6. Let A and B be two $\mathcal{L}$-fuzzy sets. Then the least relation $\perp\!\!\!\perp_{AB}$ between A and B is defined by $\perp\!\!\!\perp_{AB}(a,b) = 0$ for all $a \in A$ and $b \in B$.
The greatest relation $\top\!\!\!\top_{AB}$ between A and B is defined by $\top\!\!\!\top_{AB}(a,b) = 1$ for all $a \in A$ and $b \in B$

Definition 3.3.7. Given an $\mathcal{L}$-fuzzy relation $Y : A \to B$, for all $a \in A, b \in B$,

$$\mathbb{I}_{AB}(a,b) = \begin{cases} 1 & \text{iff } a = b \\ 0 & \text{Otherwise} \end{cases}$$

From the properties and operators in Definition 3.3.6 and Definition 3.3.7, we can derive several additional properties such as those below. Their proofs can be found in [25, 29]

Given $P, Q : A \to B$ and $R, T : B \to C$

$P;\mathbb{I}_B = P$ and $\mathbb{I}_B;T = T$

$P;\perp\!\!\!\perp = \perp\!\!\!\perp$ and $\perp\!\!\!\perp;T = \perp\!\!\!\perp$

3.3.5 The Arrows and Alpha cut Operations

The up arrow ($\uparrow$), down arrow ($\downarrow$) and α-cut are the operations we use to defuzzify fuzzy or $\mathcal{L}$-fuzzy relations to crisp relations. We use the α-cut to change the membership value of pair in an $\mathcal{L}$-fuzzy relations greater or equal to the α to degree 1. Those pairs with membership degree not greater or equal to α are set to degree 0. The effect of α-cut, which is **Thold** in $\mathcal{L}$FSQL **WHERE** clause is to put additional restriction on the condition.

Definition 3.3.8. Given an $\mathcal{L}$-fuzzy relation $R : A \times B \to \mathcal{L}$, for all $\alpha \in \mathcal{L}, a \in A$ and $b \in B, \alpha - cut$ on R i.e.

$$R_\alpha(a,b) = \begin{cases} 1 & : R(a,b) \geq \alpha \\ 0 & \text{Otherwise} \end{cases}$$

The down arrow operation works like the α-cut. In other words, it can be considered as special case of α-cut operation. The ($\downarrow$) sets all the $\mathcal{L}$ values less than 1 to 0 and the up arrow ($\uparrow$) converts all $\mathcal{L}$ values greater than 0 to 1. The up arrow is the dual of the down arrow operation. We give examples of these operations using the $\mathcal{L}$-fuzzy relation R below.

$$R = \begin{pmatrix} 1 & TI^+ & NI^+ & TI^- \\ 0 & 1 & I & TI^+ \\ NI^+ & TI^- & 1 & 0 \\ 0 & NI^+ & NI^- & 1 \\ TI^+ & 1 & 0 & I \end{pmatrix}$$

$$R_I = \begin{array}{c} \\ S1 \\ S4 \\ S3 \\ S2 \\ S5 \end{array} \begin{array}{c} \begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

$$R^{\downarrow} = \begin{array}{c} \\ s1 \\ s4 \\ s3 \\ s2 \\ s5 \end{array} \begin{array}{c} \begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

$$R^{\uparrow} = \begin{array}{c} \\ s1 \\ s4 \\ s3 \\ s2 \\ s5 \end{array} \begin{array}{c} \begin{array}{cccc} P.Sci & Comp & Geog & Maths \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

The elements of a lattice $\mathcal{L}$ can be identified with certain relations. For example, we can do so through scalars. We can use a scalar relation which is a notion introduced by Furusawa and Kawahara [15] to set $\mathcal{L}$-relations to crisp relations.

Definition 3.3.9. A relation $\alpha : A \to A$ is referred to as a scalar on A iff $\alpha \sqsubseteq \mathbb{I}_A$ and $\top_{AA}; \alpha = \alpha; \top_{AA}$.

Example 3.3.10. Let us assume that A is a set with 4 elements. Then a scalar α on R is an $\mathcal{L}$-fuzzy relation on A as in Figure 3.10 where x is an arbitrary element from $\mathcal{L}$.

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & x & 0 & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & x \end{pmatrix}$$

Figure 3.6: A scalar relation on A where x is any $\mathcal{L}$ element

The α-cut can be computed using the other operations. If α is a scalar and there exists an $\mathcal{L}$-fuzzy relation $R : A \to B$ then $(\alpha \backslash R)^{\downarrow}$ is the α-cut of R. We can view the α-cut of R if the elements of α in $\mathcal{L}$ are known.

$$(\alpha \backslash R)^{\downarrow}(a, b) = 1 \iff \alpha \leq R(x, y).$$

We compute the α-cut of R the students' records relation based on the α scalar.

$$\alpha = \begin{pmatrix} TI^{+} & 0 & 0 & 0 & 0 \\ 0 & TI^{+} & 0 & 0 & 0 \\ 0 & 0 & TI^{+} & 0 & 0 \\ 0 & 0 & 0 & TI^{+} & 0 \\ 0 & 0 & 0 & 0 & TI^{+} \end{pmatrix}$$

$$(\alpha \backslash R)^{\downarrow} = \left(\begin{pmatrix} TI^{+} & 0 & 0 & 0 & 0 \\ 0 & TI^{+} & 0 & 0 & 0 \\ 0 & 0 & TI^{+} & 0 & 0 \\ 0 & 0 & 0 & TI^{+} & 0 \\ 0 & 0 & 0 & 0 & TI^{+} \end{pmatrix} \backslash \begin{pmatrix} 1 & TI^{+} & NI^{+} & TI^{-} \\ 0 & 1 & I & TI^{+} \\ NI^{+} & TI^{-} & 1 & 0 \\ 0 & NI^{+} & NI^{-} & 1 \\ TI^{+} & 1 & 0 & I \end{pmatrix} \right)^{\downarrow}$$

$$(\alpha \backslash R) = \begin{pmatrix} 1 & 1 & NI^{+} & TI^{-} \\ 0 & 1 & NI^{+} & 1 \\ NI^{+} & TI^{-} & 1 & 0 \\ 0 & NI^{+} & NI^{-} & 1 \\ 1 & 1 & 0 & NI^{+} \end{pmatrix}$$

$$(\alpha \backslash R)^{\downarrow} = \begin{pmatrix} 1 & 1 & NI^{+} & TI^{-} \\ 0 & 1 & NI^{+} & 1 \\ NI^{+} & TI^{-} & 1 & 0 \\ 0 & NI^{+} & NI^{-} & 1 \\ 1 & 1 & 0 & NI^{+} \end{pmatrix}^{\downarrow} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

3.4 Lattice-Ordered Semigroups

3.4.1 Lattice-Ordered Semigroup

In this section, we discuss the general means to define additional operations for the logical connectors such as **AND** and **OR**. In fuzziness, t-norms and t-conorms are the very essential function that help us to get additional operators instead of the max

and the min for the logical connectors for $\mathcal{L}$-fuzzy set and relations. A general way to define these operations is through a complete lattice-ordered semigroup.

Definition 3.4.1. Let $\mathcal{L}$ be a complete bounded distributive lattice, $*$ a binary operator on $\mathcal{L}$ and $e, z \in \mathcal{L}$, then $(\mathcal{L}, *, e, z)$ is a complete lattice-ordered semigroup such that the following properties hold for all $a, b, c \in \mathcal{L}$.

$a \leq b \iff a * c \leq b * c$ Monotonicity on $*$

$a * (b * c) = (a * b) * c$ Associativity on $*$

$e * a = a$ and

$a * e = a \ \forall a \in \mathcal{L}$ Right and left neutral element on $*$ is e

$z * a = z$ and

$a * z = z \ \forall a \in \mathcal{L}$ Right and left zero element on $*$ is z

$a * \left(\bigvee_{i \in I} b_i\right) = \bigvee_{i \in I} (a * b_i)$ and

$\left(\bigvee_{i \in I} b_i\right) * a = \bigvee_{i \in I} (b_i * a)$ Distributivity of $*$

Interestingly, if $\mathcal{L}$= [0,1] and the complete lattice-ordered semigroup on $\mathcal{L}$ has 0 as the neutral element and 1 as the zero element, then the operator is known as a t-conorm. Dually iff 1 is the neutral and 0 is the zero element, then the operator is called a t-norm. Therefore, we define the following:

Definition 3.4.2. A binary operator $*$ is a t-conorm like operator iff it is an operator in a complete lattice-ordered semigroup $(\mathcal{L}, *, 0, 1)$.

Definition 3.4.3. A binary operator $*$ is a t-norm like operator iff it is an operator in a complete lattice-ordered semigroup $(\mathcal{L}, *, 1, 0)$.

More information on these operations and proofs of their properties can be found in [29].

3.4.2 Meet, Join and Composition Based on $*$

In this subsection, we define meet, join and composition based on operation $*$ on $\mathcal{L}$-fuzzy relations and give concrete examples of such definitions.

Definition 3.4.4. Given $\mathcal{L}$-fuzzy relations $R, Q : A \to B$ and $S : B \to C$ for all $a \in A, b \in B$ and $c \in C$,

$$(R \sqcap_* Q)(a,b) = R(a,b) * Q(a,b) \qquad \text{meet based on } *$$

$$(R ;_* S)(a,c) = \bigsqcup_{b \in B} \big(T(a,b) * S(b,c)\big) \qquad \text{composition on } *$$

Example 3.4.5. We continue to base this example also on the relations R and Q from the students' records. Let us define the drastic sum $*_1$ and drastic product $*_2$ as follows:

$$a *_1 b = \begin{cases} a & \text{if } b = 1 \\ b & \text{if } a = 1 \\ 0 & \text{otherwise} \end{cases} \qquad a *_2 b = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ 1 & \text{otherwise} \end{cases}$$

Then we compute the following result in our example:

$(R \sqcap_{*_1} Q)(S4, P.Sci) = R(S4, P.Sci) *_1 Q(S4, P.Sci) = 0$
$(R \sqcap_{*_2} Q)(S1, Geog) = R(S1, Geog) *_2 Q(S1, Geog) = 1$

$(R \sqcup_{*_1} Q)(S4, P.Sci) = R(S4, P.Sci) *_1 Q(S4, P.Sci) = 0$
$(R \sqcup_{*_2} Q)(S5, Comp) = R(S5, Comp) *_2 Q(S5, Comp) = 1$

$(R_{;*_2} R^T)(S1, S2) = (1 *_2 0) \vee (TI^+ *_2 1) \vee (NI^+ *_2 I) \vee (TI^- *_2 TI^+)$
$= 0 \vee 1 \vee 1 \vee 1 = 1$

3.5 Disjoint Union of Set

In crisp relational databases, an object (row) is made up of cross products of the elements of the various sets or attributes. As a matter of fact, we view a crisp relational database table as a cross product of the sets. In our $\mathcal{L}$-fuzzy database, we actually store a set as a value for an attribute. For example, storing **$Young** as a value in the field of Age in an $\mathcal{L}$-fuzzy database is actually storing the $\mathcal{L}$-fuzzy set **$Young** in the Age field.

The disjoint union of sets is defined as $A + B = \{(1,a) \mid a \in A\} \cup \{(2,b) \mid b \in B\}$. That is, it contains elements from both sets with the information from which set they belong (the tags 1 and 2). If A and B are disjoint, then $A \cup B$ is isomorphic to $A + B$,

which is essentially the same. We define $\iota(a) = (1, a)$ and $\kappa(b) = (2, b)$ as functions from A and B respectively to $A + B$. If you now consider those functions as crisp in $\mathcal{L}$-fuzzy relations, then they satisfy the properties of Lemma 3.5.1.

Lemma 3.5.1. Given $\mathcal{L}$-fuzzy sets A and B, an object $A+B$ together with additional two relations $\iota : A \to A + B$ and $\kappa : B \to A + B$ then the following properties hold:

1. $\iota; \iota^T = \mathbb{I}_A$

2. $\kappa; \kappa^T = \mathbb{I}_B$

3. $\iota; \kappa^T = \perp\!\!\!\perp_{AB}$

4. $\iota^T; \iota \sqcup \kappa^T; \kappa = \mathbb{I}_{A+B}$

We assume for simplicity, the sets involved in a disjoint union are always disjoint. Therefore, the disjoint union becomes just a union. The injections map an element from one of the two sets to itself in the union, i.e., $\iota(a) = a$ and $\kappa(b) = b$. If $D \subseteq E$ and there is a function $f : E \to C$, then we write $f|D$ for the restriction of f to D values only, i.e. $f|D : D \to C$ and $(f|D(a)) = f(a)$. Using the convention that both sets in a disjoint union are disjoint, we have $A \subseteq A + B$ so that $f|A$ is properly defined for every $f : A + B \to C$.

As we have mentioned earlier that a value to be stored in an $\mathcal{L}$-fuzzy relational database is an $\mathcal{L}$-fuzzy set. Therefore, a table of a database with n rows and 3 attributes with domains A, B and C respectively, becomes a crisp function $f : n \to L^A \times L^B \times L^C$ where n is a set with n elements. Since we have that $\mathcal{L}^A \times \mathcal{L}^B \times \mathcal{L}^C$ is isomorphic to $\mathcal{L}^{A+B+C}$, we can model the table alternatively by a crisp function $f' : n \to \mathcal{L}^{A+B+C}$. Finally, any crisp function of this type corresponds to exactly one $\mathcal{L}$-fuzzy relation $R : n \to A + B + C$. Indeed, the relationship between f and R is given by $R(m, x) = l \iff f(m)(x) = l$.

Chapter 4

$\mathcal{L}$FSQL and $\mathcal{L}$-Fuzzy Databases

In this chapter, we define and discuss the $\mathcal{L}$-fuzzy database and its formal language $\mathcal{L}$FSQL. This work is inspired by the works in [6, 7, 8, 28]. We will first look at the elements in the $\mathcal{L}$-fuzzy databases. We will proceed to look at $\mathcal{L}$FSQL syntax, the grammar and conclude with the semantics of $\mathcal{L}$FSQL.

4.1 $\mathcal{L}$-Fuzzy Databases

This section discusses $\mathcal{L}$-fuzzy database tables. It also covers the meta-knowledge database that stores the lattice $\mathcal{L}$, semigroup on $\mathcal{L}$ and membership function for certain $\mathcal{L}$-fuzzy sets. We continue the discussion to cover the nature of $\mathcal{L}$-fuzzy data we store in the tables. We prefixed linguistic labels with \$ to help distinguish from other values as was done in [6]. In the case of a function for $\mathcal{L}$-fuzzy sets, we will prefix with #.

4.1.1 Tables and Meta-Knowledge Base

As much have been said in Chapter 2 about regular relational databases, $\mathcal{L}$-fuzzy databases also consist of tables which are made up of columns and rows. The basic rules in regular relational databases also apply to $\mathcal{L}$-fuzzy databases. For example, a table name should be unique as well as the attribute names in a table. An attribute has a domain which determines the set of specific kinds of values to be stored in it. Attributes of $\mathcal{L}$-fuzzy databases allow $\mathcal{L}$-fuzzy data to be stored in them. The domain on these $\mathcal{L}$-fuzzy sets can be linearly ordered or partially ordered

The $\mathcal{L}$-fuzzy databases also contain a meta-knowledge base which stores the operations

such as t-norm and t-conorm like operations, linguistic labels and lattice $\mathcal{L}$. Linguistic labels are commonly used imprecise expressions such as old and young that we extend to derive an $\mathcal{L}$-fuzzy set. In the meta-knowledge base of the $\mathcal{L}$-fuzzy database, we can explicitly list the $\mathcal{L}$-fuzzy set such as **$Young** = {1/15yrs,..., 1/2yrs5, γ/26yrs, ..., γ/29yrs, α/30yrs, ..., α/39yrs, 0/40yrs, ...}. The meta database stores a function f. In $\mathcal{L}$FSQL we refer to such a function by $(\#f)$. We can define a membership function for **$Young** as:

$$\textbf{\$Young}(x) = \begin{cases} 1 & \text{iff } x \leq 25\text{yrs} , \\ \gamma & \text{iff } 25\text{yrs} < x \leq 29\text{yrs} ; \\ \alpha & \text{iff } 29\text{yrs} < x \leq 39\text{yrs} ; \\ 0 & \text{otherwise} \end{cases}$$

One important notion is that each domain allows a set of binary comparison operators to be used on them. An ordered domain allows equalities and inequalities comparators whilst unordered domain allows at least equality comparators. For example, in an ordered domain, we can use "$\geq$" on values a and b as $a \geq b$ iff $a > b$ or $a = b$. Some domains also allow approximate equality ($\equiv$). The approximate equality returns the degree to which two values are almost equal or differ from each other which is an $\mathcal{L}$-fuzzy relation. We have to note that the approximate equality cannot be transitive but must be reflexive and symmetric i.e., $a \equiv a = 1$ and $a \equiv b = b \equiv a$ respectively [9].

4.1.2 $\mathcal{L}$-Fuzzy Data

Generally, all data in the $\mathcal{L}$-fuzzy database are $\mathcal{L}$-fuzzy sets in nature. In other words, every entry in an $\mathcal{L}$-fuzzy database table is an $\mathcal{L}$-fuzzy set. We consider crisp values as a special case of $\mathcal{L}$-fuzzy set. In other words, it is a fuzzy set with only one value in it. A crisp value is modeled by having the degree of the corresponding fuzzy value set to 1 and the rest set to 0. If we write a value such as 18 for age in an $\mathcal{L}$-fuzzy database, the 18 is a short hand of {1/18}. Generally speaking, if we write a set such as {5,20,24} as a value, we really mean that all the elements have degree 1, i.e. {1/5,1/20,1/24}.

We can store linguistic labels directly or explicitly in $\mathcal{L}$-fuzzy database attributes. For example we can store **$Young** or **$Old** or **$Very_Old** in the Age field of $\mathcal{L}$-fuzzy database. These labels are just a name referencing an $\mathcal{L}$-fuzzy set already stored

in the meta-knowledge base. We can also define a new $\mathcal{L}$-fuzzy set as a subset of a predefined $\mathcal{L}$-fuzzy set in the meta-knowledge base. Conversely, we can combine several predefined fuzzy sets to form a new $\mathcal{L}$-fuzzy set.

Another possible way to obtain a new $\mathcal{L}$-fuzzy set is by computing the lower and the upper bounds of an already existing $\mathcal{L}$-fuzzy set. Let us assume that s is an $\mathcal{L}$-fuzzy set, then $lbd(s)$ and $ubd(s)$ represents the lower bound and upper bounds of s respectively. We compute the lower and the upper bounds on ordered domains. For example, in Figure 4.1, we have used $\mathcal{L}$=[0..1] and obtained **$Young** from **$Old** using the lower bound operation, which is **$Young**=$lbd$(**$Old**). Similarly, we have **$Very_Old**=$ubd$(**$Old**). We can also modify the lower and upper bounds computation by using the t-norm and the t-conorm like operations such as $lbd(*, s)$ and $ubd(*, s)$ respectively.

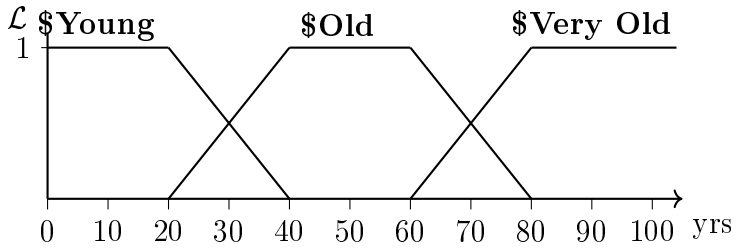


Figure 4.1: Age as a fuzzy set

We can also obtain a new set through approximate equality as discussed in Section 4.1.1. The approximate equality operations serve as a means to strengthening or loosening the notation given by the set. We use $extremely(\equiv, s)$ and $very(\equiv, s)$ to denote the strengthening modifiers on s. On the other hand, $more\_or\_less(\equiv, s)$ and $roughly(\equiv, s)$ are also used as loosening modifiers on s. For example, in **$Young**, we might consider two individual with ages differ by 2 months as extremely equal or those differ by 6 months as very equal and so on.

4.2 $\mathcal{L}$FSQL

In this section, we discuss the formal language $\mathcal{L}$FSQL and $\mathcal{L}$-fuzzy databases. We will focus our discussion on the create, insert, delete and the select statements only. Others like the update can be done using those arrow operations. We will start by looking into some of the basic elements that forms these $\mathcal{L}$FSQL statements. In our discussion of some of these elements, we will provide part of the grammar which

relates to them to aid the reader's understanding. We will use Extended Backus Naur Form (EBNF) grammar as a formal notation to describe the $\mathcal{L}$FSQL. Following the symbols and their interpretations in the EBNF, we use angle brackets ($< >$) to enclose expression and square bracket ([ ]) for optional. We also use curly brackets ({}) for repetition and vertical bar or stroke (|) as alternative. The expressions are at the left hand side whilst their interpretation are on the right hand side of the symbol (::=). Those names which are not in the angle brackets are the terminals. More information on EBNF can be found in [24]. For the full grammar, we refer the reader to Section 4.3. This $\mathcal{L}$FSQL language is inspired by the works in [6, 7, 8, 28].

4.2.1 Elements in $\mathcal{L}$FSQL

We discuss some of the basic elements that form the language $\mathcal{L}$FSQL.

$\mathcal{L}$-Fuzzy Comparators

Comparators are used in the **WHERE** clause to compare an $\mathcal{L}$-fuzzy value with an attribute or to compare attributes of the same kind. The comparators used in $\mathcal{L}$FSQL are possibility and necessity comparators. We follow the same notation in [6]. In $\mathcal{L}$FSQL, as we know from Subsection 3.4.3, we can store $\mathcal{L}$-fuzzy sets as a value in a field. Now to compare in $\mathcal{L}$FSQL, we extend from comparing elements in a set to comparing $\mathcal{L}$-fuzzy sets. Let us denote the letter "C" to represent a binary comparator. We assume that a comparison in $\mathcal{L}$FSQL is syntactically correct if the attribute and the value or the other attribute are of the same kind.

Possibility (Fuzzy) Comparators (FC) are less restrictive, they use general features to compare elements [6]. In fact they select more tuples than the necessity comparators. Mathematically, the possibility comparators use composition discussed in Subsection 3.3.3 to operate. An $\mathcal{L}$FSQL query which makes use of the possibility comparator in the **WHERE** clause seeks to select tuples with least satisfaction to the condition. For example, let us assume that **$Heavy** = $\{0/45kg,..,0/64kg,\ \beta/65kg,..,\ \beta/70kg,\ \delta/71kg,\ ..,\delta/80kg,\ 1/81kg,..\}$ and we don't know the weight of Eric. We are certain that he will be $69kg$, $70kg$ or $71kg$. If we want to compare using FC whether Eric's weight is **$Heavy** with degree δ, the response will be true. Despite that $69kg$ and $70kg$ do not satisfy the condition, $71kg$ provides the needed least satisfaction to the condition. Some of the examples of the FC are Fuzzy equal (**F**= or **FEQ**), Fuzzy

less or equal(**F**≤ or **FLEQ**) and Fuzzy Greater Than (**F**> or **FGT**)

Necessity (Fuzzy) Comparators (NFC) on the other hand are more restrictive. They select tuples that satisfy the condition fully. A positive result from a condition with the NFC means that all the specified features are satisfied fully. Due to the nature of NFC, it selects less tuples as compared to FC when used in $\mathcal{L}$FSQL select statement. Mathematically, the Necessity comparators use a residual discussed in Subsection 3.3.3 to operate. In the case of Eric's weight as an example in the FC paragraph, if we use the NFC instead of FC the result will be false. The weights $69kg$ and $70kg$ do not have the degree δ as is required by the condition. An example of NFC is Necessity Fuzzy equal (**NF**= or **NFEQ**). More examples of the comparators can be found in [6].

Below is the part of the $\mathcal{L}$FSQL grammar which represents $\mathcal{L}$-fuzzy comparators.

```
<F_comparator> ::=  <F_equal> | <NF_equal>
                    | <F_different > | <NF_different>
                    | <F_GreaterThan>
                    | <NF_GreaterThan>
                    | <F_GreatThanOrEqual>
                    | <NF_GreatThanOrEqual>
                    | <F_LessThan>| <NF_LessThan>
                    | <F_LessThanOrEqual>
                    | <NF_LessThanOrEqual>

<F_equal> ::= FEQ | F''=''
<NF_equal> ::= NFEQ | NF''=''
<F_different >  ::= FDIF | F''!='' | F''<>''
<NF_different >  ::= NFDIF | NF''!='' | NF''<>''
<F_GreaterThan> ::= FGT | F''>''
<NF_GreaterThan> ::= NFGT | NF''>''
<F_GreatThanOrEqual> ::= FGEQ | F''>=''
<NF_GreatThanOrEqual> ::= NFGEQ | NF''>=''
<F_LessThan> ::= FLT | F''<''
<NF_LessThan> ::= NFLT | NF''<''
<F_LessThan OrEqual> ::= FLEQ | F''<=''
<NF_LessThan OrEqual> ::= NFLEQ | NF''<=''
```

$\mathcal{L}$-Fuzzy Threshold

The threshold (**THOLD**), as we have mentioned earlier in this work, is the α-cut which puts additional restriction on the condition. It represents the degree to which a condition needs to be satisfied. The **THOLD** comes with its degree. For example, in the condition

Height **F>** **$Tall** **THOLD** α,

α is the thold degree posing restriction on the condition. The fulfillment threshold is optional in $\mathcal{L}$FSQL. In other words, we can write an $\mathcal{L}$FSQL condition without a restriction from the fulfillment threshold if we don't add it.

$\mathcal{L}$-Fuzzy Logical Connectors

The logical connectors are used to connect conditions in $\mathcal{L}$FSQL queries. We will use them to form multiple conditions in the **WHERE** clause as was done in FSQL [6]. Similarly to the crisp and fuzzy relational database logical connectors, we use **AND** and **OR** which represent meet and join operators of the lattice $\mathcal{L}$ respectively. Instead of using by default the meet and join operations for **AND** and **OR** respectively, we can specify t-norm like and t-conorm like operators such as **AND(*)** and **OR(*)**. We obtained these t-conorm and t-norn like operators from lattice-ordered semigroups discussed in Section 3.4. When two conditions are connected by a logical connector, both conditions are executed before the logical connector is applied to the two results. For example in the statement

Height **FGT** **$Very_Tall** **THOLD** γ **AND(*)** Weight **NFLEQ** **$Heavy** **Thold** β,

the conditions for height and weight will be executed first, then the * operation is used on the results from the two conditions.

4.2.2 $\mathcal{L}$FSQL Statements

In this subsection, we now consider discussing the $\mathcal{L}$FSQL **SELECT**, **DELETE**, **INSERT** and **CREATE** statements which are formed from the elements discussed so far in this section.

CREATE Statement

The **CREATE** statement is used to create a new empty table from the attributes specified in the statement. It contains the table name, the set of attributes names and the domains of the attributes. The syntax for a **CREATE** statement is:

$$\textbf{CREATE TABLE } T(A_1 : D_1, A_2 : D_2, ..., A_n : D_n);$$

where T is the table name, $A_1, A_2, ..., A_n$ are the attributes names and $D_1, D_2, ..., D_n$ are the corresponding domains of the attributes. For the **CREATE** statement to be syntactically correct, the table name must be unique in the database. In addition, attributes names should be unique in their local table and there has to be a domain for each attribute.

The grammar giving a pictorial view of the $\mathcal{L}$FSQL **CREATE** statement is

```
<Create> ::= CREATE table_name "("<field>
             [{"," <field> }]")" ";"
<field> ::= attribute <data_type>
<data_type> ::= INTEGER | DOUBLE | STRING | CHAR | BOOL
```

INSERT Statement

The **INSERT** statement is used to add new tuples to the existing table. We state the table to insert into, the attributes in the table and the $\mathcal{L}$-fuzzy values. Syntactically an $\mathcal{L}$FSQL **INSERT** query will be:

$$\textbf{INSERT INTO } T \ (A_1, A_2, ..., A_n) \textbf{ VALUES } (V_1, V_2, ...V_n);$$

where T is the table to insert the new tuples into, $A_1, A_2, ..., A_n$ are the attributes in the table T. The $V_1, V_2, ...V_n$ on the other hand are the $\mathcal{L}$-fuzzy values from the domain $D_1, D_2, ..., D_n$ corresponding to the attributes $A_1, A_2, ..., A_n$. As we already know, we can explicitly insert $\mathcal{L}$-fuzzy values directly into the table. The **INSERT** statement is syntactically correct if the values are in the domain of their corresponding attributes.

The SELECT Statement

The **SELECT** statement is used to retrieve tuples that satisfy a given condition. In other words, we use it to create new table from existing tables based on a given

condition. It is made up of the project part i.e. **SELECT** clause, the **FROM** clause and the condition part which is the **WHERE** clause. Syntactically, an $\mathcal{L}$FSQL **SELECT** statement will be:

SELECT $A_1, A_2, ..., A_n$ **FROM** $T_1, T_2, ...T_n$
WHERE Y $\mathcal{L}FC$ Y' **Thold** $\mathcal{L}$;.

The $A_1, A_2, ..., A_n$ represent the attributes, $T_1, T_2, ...T_n$ represent the tables and Y, Y' represent attributes of the same kind or an attribute and an $\mathcal{L}$-fuzzy value. Lastly $\mathcal{L}$FC on the other hand represents one of the $\mathcal{L}$-fuzzy comparators discussed in this section.

The **SELECT** clause or project part determines the tuples to be selected if a given condition is satisfied. We list the attributes from which we project the resulting tuples. An example will be:

SELECT Id, Height, Weight, ...

The **FROM** clause works similarly to the classical relational database. We state the table or tables we want to select the tuples from.

The **WHERE** clause is the most complex structure as compared to the **SELECT** and the **FROM** clauses of a query. A simple **WHERE** clause contains attributes or an attribute and an $\mathcal{L}$-fuzzy value to be compared by an $\mathcal{L}$-fuzzy comparator. There can also optionally be a **THOLD** with its degree as well. The **WHERE** clause is considered to be syntactically correct if the comparing attributes and the $\mathcal{L}$-fuzzy comparator are of the same domain. Due to the logical connectors discussed in this section, the **WHERE** clause accommodates multiple conditions. An example of **WHERE** clause could be:

WHERE Height **NF**$>$ **$Very_Tall THOLD** β
AND(*) Weight **FLEQ $Heavy**;

Below is a part of $\mathcal{L}$FSQL grammar providing overview of the **WHERE** clause.

```
<where_clause>::= WHERE <F_condition_Form>
<F_condition_Form> ::= <F_condition>
                       [{<L_operator> <F_condition>}]
<F_condition> ::= <compare_value> <F_comparator>
                  <compare_value> [<threshold>]
```

DELETE Statement

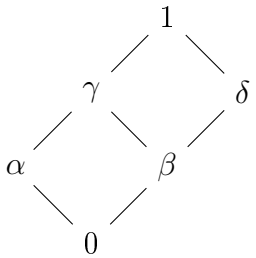
The **DELETE** statement is used to remove some tuples which satisfy the given condition in the **WHERE** clause. Syntactically, a simple **DELETE** statement will be:

DELETE FROM T **WHERE** Y $\mathcal{L}FC$ Y' **Thold** $\mathcal{L}$;

To delete all the tuples from a table, we drop the **WHERE** clause in the **DELETE** statement. The **DELETE** statement is syntactically correct if the table in the query exists.

Example 4.2.1. Let us consider creating a single table in an $\mathcal{L}$-fuzzy database to store records of basketball players. We will use $\mathcal{L}$FSQL statements discussed above to interact with the table. In other words, we will use $\mathcal{L}$FSQL **SELECT**, **DELETE**, **INSERT** and **CREATE** statements to demonstrate what we have discussed so far. In this example, we will use the lattice structure from Figure 3.4 repeated below for easy understanding of the reader.

Let us assume that $\mathcal{L}$-fuzzy sets **$Tall** and **$Heavy** based on Figure 3.4 are stored in the meta-knowledge base. **$Tall** and **$Heavy** are measured in feet and kilogram respectively.
$Tall =$\{0/5.0, .., 0/5.4, \alpha/5.5 ...., \alpha/5.9, \gamma/6.0, .., \gamma/6.4, 1/6.5,....\}$
$Heavy = $\{0/45, .., 0/64, \beta/65, .., \beta/70, \delta/71, .., \delta/80, 1/81, ...\}$



We create the table Players to store records of basketball players by the $\mathcal{L}$FSQL **CREATE** statement. We want to store data on a player's unique number (Player_Id), the name of the player (Name) and years for his contract(Contract). We will consider these attributes to hold crisp values. We also want to store the player's height (Height) and the weight (Weight) as at signing of the contract. We will assume that Height and Weight attributes take an $\mathcal{L}$-fuzzy values.

CREATE TABLE Players (Player_Id String, Name String, Height Float, Weight Float, Contract Int);

From the **CREATE** statement, we can see that the table name is Players. In addition, every attribute comes with a domain. To be more precise Contract has an integer domain, Player_Id and Name have string and both Height and Weight have float domain. Table 4.1 is the resulting table from the **CREATE** statement. The table is empty after the **CREATE** statement. The attributes specified in the statement have formed the headings for the columns.

Player_Id	Name	Height(ft)	Weight (kg)	Contract(yr)

Table 4.1: The resulting table from the **CREATE** query

We populate the Players' table with 5 records using the **INSERT** statement. We will only write the **INSERT** statement of the first record in this paper. The **INSERT** statements for the remaining records in Table 4.2 follow similarly **INSERT** statement as the first one. The **INSERT** statement will be:

INSERT INTO Players (Player_Id, Name, Height, Weight, Contract) **VALUES** (P10, Kelly, **$Tall**, 75, 5);

In the **INSERT** statement, we specified the table name which is Players, the attributes and their corresponding values. Table 4.2 is the resulting table of the **INSERT** queries. As we have discussed earlier on in this section, we have been able to insert $\mathcal{L}$-fuzzy values **$Tall** and **$Heavy** in the table. We can also observe that the table has been able to accommodate an explicitly list $\mathcal{L}$-fuzzy set in a cell which is {5.5,6}. Again, the values such as Kelly, Thomas, 75 and 5.2 are abbreviations of the $\mathcal{L}$-fuzzy sets in which the corresponding value has degree 1 and all other values have degree 0.

Player_Id	Name	Height(ft)	Weight (kg)	Contract(yrs)
P10	Kelly	$Tall	75	5
P08	Thomas	5.2	$Heavy	2
P02	John	{5.5,6}	79	4

Table 4.2: Resulting table of **INSERT** statement

The next statement is to retrieve some records from Table 4.2. using the **SELECT**

statement. We specified Player Id, Name, Height and Weight as the attributes to form the resulting table if the given condition is satisfied. We are selecting from the Players table. In the **WHERE** clause, we want to restrict and pick only players who are fuzzy equal to **$Tall** with degree γ and also necessary less or equal to **$Heavy** with minimum degree δ. We used the **SELECT** query:

SELECT Player_Id, Name, Height, Weight **FROM** Players **WHERE** Height F= **$Tall THOLD** γ **AND** Weight NF$\leq$ **$Heavy THOLD** δ;

Table 4.3 is the resulting table for the **SELECT** query. We can observe that contract is not part of the resulting table because we did not state it at the **SELECT** clause. If we consider the records in Table 4.3, Kelly's height, **$Tall** has a degree 1 and his weight 75 has a degree δ in the $\mathcal{L}$-fuzzy set **$Heavy**. Both degrees make Kelly satisfy the given conditions in the **WHERE** clause, therefore the row containing Kelly is selected. In the case of John, his height value 5.5 has a degree α which is not up to the threshold γ. However since we are using a possibility comparator and his other height value 6 has degree γ, he qualifies to be part of the resulting table. Thomas on the other hand does not satisfy both conditions so he has been excluded from the resulting table.

Player_Id	Name	Height(ft)	Weight (kg)
P10	Kelly	$Tall	75
P02	John	{5.5,6}	79

Table 4.3: Resulting table from the select query

In the **DELETE** statement below, we want to delete the records with weight fuzzy greater or equal to $Heavy from Table 4.2. From the **DELETE** statement, we want to delete only those satisfying the condition with a degree of 1. Due to this we have omitted both the threshold and its degree. The **DELETE** statement will be:

DELETE FROM Player **WHERE** Weight F$\geq$ $Heavy δ;

The resulting table from the **DELETE** statement is Table 4.4. The interpretation of the condition in the **WHERE** clause follows the same process explained in the **SELECT** statement of this subsection. It is only Thomas who satisfies the given condition and his record did not appear in Table 4.4.

Player_Id	Name	Height(ft)	Weight (kg)	Contract
P10	Kelly	$Tall	75	7
P02	John	{5.5,6}	79	5

Table 4.4: Resulting table of insert statement

4.3 The Grammar for ℒFSQL

In this section, we basically provide the full grammar including some part that we have not introduced in our discussions so far.

ℒFSQL STATEMENT

```
<LFSqlStatement> ::= <Create> | <Insert> | <Select> | <Delete>
```

Create statement grammar

```
<Create> ::= CREATE table_name "("<field>
             [{"," <field> }]")" ";"
<field> ::= attribute <data_type>
<data_type> ::= INTEGER | DOUBLE | STRING | CHAR
```

Insert statement grammar

```
<Insert> ::= INSERT INTO <table_name> VALUES "("<fuzzy_value>
             [{"," <fuzzy_value>}]")" ";"

<fuzzy_value> ::= <“$” label> | <explicit_value>
<explicit_value>::= “{” <degree>"/"<valu>
                    [“,” <degree>"/"<value> ]”}”

 <degree> :: = lattice
 <value> :: = integer | float | string | char
```

Select statement grammar

```
<Select> ::= SELECT <select list> <from_clause>
             [<where_clause>]“;”

<select_sublist> ::= <attribute> [{"," <attribute>}]
                    |<qualified_Attributes>
                    [{"," <qualified_Attributes>}]
```

```
<qualified_attribute> ::= table_name.attribute

<from_clause> ::= FROM <table_clause>

<table_clause>::= table_name [{'',''table_name [aliase_name]}]

<where_clause>::= WHERE <F_condition_Form>
<F_condition_Form> ::= <F_condition>
                       [{<L_operator> <F_condition>}]

<F_condition> ::= <compare_value> <F_comparator>
                  <compare_value> [<threshold>]

<compare_value>::= <fuzzy_value>|<qualified_attribute>
                   | attribute

<threshold> ::= THOLD <degree>
<degree>::= lattice_value

<L_operator> ::= AND | OR

<F_comparator> ::=  <F_equal> | <NF_equal>
                   | <F_different > | <NF_different>
                   | <F_GreaterThan>
                   | <NF_GreaterThan>
                   | <F_GreatThanOrEqual>
                   | <NF_GreatThanOrEqual>
                   | <F_LessThan>| <NF_LessThan>
                   | <F_LessThanOrEqual>
                   | <NF_LessThanOrEqual>

<F_equal> ::= FEQ | F''=''
<NF_equal> ::= NFEQ | NF''=''
<F_different >  ::= FDIF | F''!='' | F''<>''
<NF_different >  ::= NFDIF | NF''!='' | NF''<>''
<F_GreaterThan> ::= FGT | F''>''
```

```
<NF_GreaterThan> ::= NFGT | NF''>''
<F_GreatThanOrEqual> ::= FGEQ | F''>=''
<NF_GreatThanOrEqual> ::= NFGEQ | NF''>=''
<F_LessThan> ::= FLT | F''<''
<NF_LessThan> ::= NFLT | NF''<''
<F_LessThan OrEqual> ::= FLEQ | F''<=''
<NF_LessThan OrEqual> ::= NFLEQ | NF''<=''
```

Delete statement grammar

```
<Delete> ::= DELETE FROM table_name [<where_clause>];
```

4.4 The Semantics of the $\mathcal{L}$FSQL

This section contains the semantics underlying $\mathcal{L}$FSQL. We will use $[\![\ ]\!]$ and I to mean semantics and interpretation respectively. We make the following assumptions to help us defining the semantics of $\mathcal{L}$FSQL.

1 If D is an ordered, then $(\leq)$ is a crisp relation on D.

2 If the domain D has an approximate equality, then $(\equiv)$ is an $\mathcal{L}$-fuzzy relation on D.

In order to model the part of the meta-knowledge base that stores predefined $\mathcal{L}$-fuzzy sets, we use a function σ_s that maps names or linguistic entities to $\mathcal{L}$-fuzzy sets. That is, $\sigma_s(\$m) : D \to \mathcal{L}$ where $\$m$ is the name of an $\mathcal{L}$-fuzzy subset of the domain D.

4.4.1 The Semantics of $\mathcal{L}$-fuzzy sets

As we can recall from the Section 4.1.2 all data in an $\mathcal{L}$-fuzzy database are $\mathcal{L}$-fuzzy sets and every $\mathcal{L}$-fuzzy set has a domain. We provide the semantics of $\mathcal{L}$-fuzzy sets in this subsection.

If m denotes an $\mathcal{L}$-fuzzy subset of D, then the semantics of m is an $\mathcal{L}$-fuzzy subset, which is $[\![m]\!](\sigma_s) : D \to \mathcal{L}$. Depending on m the semantics is defined as follows:

> The semantics of the term $\$m$ for a name m is simply the set that has been stored in the meta database for m. That is

$[\![\$m]\!](\sigma_s) = \sigma_s(\$m)$.

In the case of explicitly listing the elements of m together with their degree of membership, we can easily generate the corresponding $\mathcal{L}$-fuzzy subset in $D \to \mathcal{L}$. Below is the semantic representing such a situation.

$$[\![\{l_1/d_1, ..., l_n/d_n\}]\!](\sigma_s)(d) = \begin{cases} l_1 & \text{iff } d = d_1 \\ l_2 & \text{iff } d = d_2 \\ : & : \\ 0 & \text{otherwise} \end{cases}$$

If m is given by an implemented membership function stored in the meta-knowledge base, then the semantics of the function $\#f$ will be f. That is

$[\![\#f]\!](\sigma_s) = f$.

For the semantics of the approximate equalities, we give the semantics component wise. For example, the semantics of very approximately equal to m will be the residual of the semantics of m and the approximate equality of similar elements in m from the domain D . In other words, we take the meet of all the relative psuedo-complements of d and d' and the semantics of m. Below is such a representation.

$$[\![\text{ very } (\equiv, m)]\!](\sigma_s)(d) = \bigsqcap_{d' \in D} (d \equiv d') \to [\![m]\!](\sigma_s)(d').$$

In the case of extremely approximately equal to m, we take the residual operation again on the results of semantics of the very approximately equal to m above and the interpretation of the approximately equal of elements from D. We can view the semantics as

$$[\![\text{ extremely } (\equiv, m)]\!](\sigma_s)(d) = \bigsqcap_{d' \in D} (d \equiv d') \to [\![\text{very } (\equiv, m)]\!](\sigma_s)(d')$$
$$= \bigsqcap_{d' \in D} (d \equiv d') \to \Big(\bigsqcap_{d'' \in D} (d' \equiv d'') \to [\![m]\!](\sigma_s)(d'') \Big).$$

For the semantics of more_or_less approximately equal to m, we rather used the composition to give their semantics. That is, the semantics of more_or_less approximately equal to m is the join of all the degrees from the meet of any value belonging to m and approximately equal in m. We can view the semantics as

$$[\![\text{ more\_or\_less } (\equiv, m)]\!](\sigma_s)(d) = \bigsqcup_{d' \in D} [\![m]\!](\sigma_s)(d') \sqcap (d \equiv d').$$

Similarly to extremely approximately equal to m, the semantics of roughly ap-

proximately equal to m is achieved by compose again the results from more_or_less approximately equal to m. We can view it as

$[\![\text{ roughly } (\equiv, m)]\!](\sigma_s)(d) = \bigsqcup_{d' \in D} [\![\text{ more\_or\_less } (\equiv, m)]\!](\sigma_s)(d') \sqcap (d \equiv d')$
$= \bigsqcup_{(d', d'') \in D} [\![m]\!](\sigma_s)(d'') \sqcap (d \equiv d') \sqcap (d' \equiv d'')$

For the semantics of the lower bounds i.e. $lbd(m)$, we take the residual of the semantics of m and the smaller or equal of the elements in the domain D. Below is the semantics of the lower bonds.

$[\![lbd(m)]\!](\sigma_s)(d) = \bigsqcap_{d' \in D} [\![m]\!](\sigma_s)(d') \rightarrow (d \leq d')$.

In the case of upper bounds, we take the residual of the semantics of m and the smaller or equal of the elements in the domain D as was done in semantics of lower bounds. The only difference here is we have $d' \leq d$ which is contrary to the semantics of lower bounds. Below is the semantics of the upper bounds.
$[\![ubd(m)]\!](\sigma_s)(d) = \bigsqcap_{d' \in D} [\![m]\!](\sigma_s)(d') \rightarrow (d' \leq d)$.

The semantics of Tables

In reference to Chapter 3, and especially Section 3.5., we can define the semantics of a table R with n rows and $A_1, A_2, \ldots, A_n$ attributes as an $\mathcal{L}$-fuzzy relation:
$R : n \rightarrow D_1 + D_2 + \cdots + D_m$ where D_j is domain of A_j

Let us denote σ_t as a function used to map the table names to the actual object in the database. This implies that:

$\sigma_t(R) : n \rightarrow D_1 + D_2 + \cdots + D_m$
provided R is the name of the table as described above.

To insert into a table, we are changing the table. This can be viewed as an update. For example, to put V for Y in table R results R if $Y = V$ else is $\sigma_t(Y)$. Mathematical representation of this example is:

$$\sigma_t[X/R](Y) = \begin{cases} R & : X = Y \\ \sigma_t(Y) & \text{otherwise} \end{cases}$$

To compare attributes, we select the corresponding rows and columns and compare them. In order to obtain the value at certain attribute A_j from the i-th row of a table R, we use the injection from D_j into the $D_1 + ... + D_m$. That is we take the

table, obtain the i^{th} row, take the part in j^{th} column and pick the value there. The semantics of such a process is:

$[\![R.A_j]\!](\sigma_s, \sigma_t)(i) = \sigma_t(R)(i)|D_j$.

The Semantics $\mathcal{L}$-Fuzzy Comparators

The necessity and the possibility comparators use residual and composition respectively. The comparison of lattice values is achieved by taking the two values from the same domain which is either ordered or unordered domain. We then take the meet of their membership values and then compute the union of all the outcomes of the meets. The semantics of comparing two values S and S' using the possibility comparator FC is:

$$[\![SFCS']\!](\sigma_s, \sigma_t) = \bigsqcup_{d,d' \in D} \big([\![S]\!](\sigma_s, \sigma_t)(d) \sqcap FC(d, d') \sqcap [\![S']\!](\sigma_s, \sigma_t)(d')\big).$$

In the case of necessity comparators, we follow similar process as the possibility comparators discussed above. The only difference is we use the implication or relative pseudo-complement instead of meet, we then take the meet of all the outcomes from the relative pseudo-complement of the membership degrees involved. The semantics of comparing two values S and S' using the necessity comparator NFC is:

$$[\![SNFCS']\!](\sigma_s, \sigma_t) = \bigsqcap_{d,d' \in D} \big([\![S]\!](\sigma_s, \sigma_t)(d) \rightarrow NFC(d, d') \sqcap [\![S']\!](\sigma_s, \sigma_t)(d')\big)$$

The Semantics Threshold and Logical Comparators

The semantics of a threshold is defined by an α-cut as described in Section 4.2.1 and visualized in Example 3.3.10. Let us assume that Con represents an $\mathcal{L}$-fuzzy condition, then:

$$[\![Con\ THOLD\ l]\!](\sigma_s, \sigma_t) = \begin{cases} 1 & : l \leq [\![Con]\!](\sigma_s, \sigma_t) \\ 0 & \text{otherwise.} \end{cases}$$

Notice that the semantics above can also be computed by using scalar relations and the arrow operations as demonstrated in the Chapter 2.

In the case of one of the two logical connectives **AND** and **OR**, we use meet and join respectively as we have discussed in Section 4.2.1. If a t-norm or t-conorm like

operator is used in the logical connectives, then we use $\wedge_*$ or $\vee_*$ respectively instead of $\wedge$ or $\vee$.

4.4.2 The Semantics of $\mathcal{L}$FSQL Statements

In this section, we are now going to provide the semantics of the basic $\mathcal{L}$FSQL statements. We will use the elements and semantics as defined above in this section to give the semantics of these $\mathcal{L}$FSQL statements.

The Semantics of the Create Statement.

We can recall that the **CREATE** statement creates an empty table with the attributes specified in the statement. The empty table has no rows. Seen as a relation, it is the unique relation with the empty set as source. Because of the correspondence between sets of pairs and relations, we denote this relation simply by $\varnothing$. We define the semantics of a **CREATE** statement as

$$[\![\textbf{ CREATE TABLE } T(A_1 : D_1, ..., A_n : D_n)]\!](\sigma_s, \sigma_t) = \varnothing$$

The Semantics of the Insert Statement

An **INSERT** statement adds one new row to those in the table. All rows that have already been in the table remain unchanged. Insert is the union of the already inserted rows and a new row. We define the semantics of an **INSERT** statement as

$$[\![\textbf{ INSERT INTO } T \textbf{ VALUES}(v_1, .., v_k)]\!](\sigma_s, \sigma_t)(i)(d)$$
$$= \begin{cases} [\![T]\!](\sigma_s, \sigma_t)(i)(d) & : i \leq n \\ [\![v_j]\!](\sigma_s)(d) & : i = n + 1 \text{ and } d \in D_j \end{cases}$$

The Semantics of the Select Statement

Let M be a subset of the indices 1 to n i.e., $M \subseteq \{1, ..., n\}$ so that the**WHERE** clause W is satisfied. In other words, $j \in M$ iff $[\![T]\!](\sigma_s, \sigma_t)(j) \neq 0$. The semantics of selecting from table T is basically picking all the rows restricted to M. In other words, we select all rows that return true for the **WHERE** condition. The semantics of a **SELECT** statement is simply

$[\![$ **Select** $A_1, ...A_n$ **FROM** T **WHERE** Con **THOLD** l $]\!](\sigma_s, \sigma_t)$
$=[\![T]\!](\sigma_s, \sigma_t)|m$

The Semantics of Delete Statement

The **DELETE** statement is just splitting a table into two and retaining those that do not satisfy the given condition. Similar to select, the semantics of deleting from T is to select all the rows restricted to M stated above and remove them. To be more precise, we select those rows which satisfy the condition and drop them. We define:

$[\![$ **DELETE FROM** T **WHERE** Con **THOLD** l $]\!](\sigma_s, \sigma_t) = [\![T]\!](\sigma_s, \sigma_t)|m$

Chapter 5

The Implementation For $\mathcal{L}$FSQL and $\mathcal{L}$-Fuzzy Databases

In this chapter we will discuss the underlying implementations of the $\mathcal{L}$FSQL. We will first of all talk about the programming language used namely Haskell [12, 19, 22, 27]. We will focus the discussion of the language on the Parsec library [13, 16] used in the implementation of $\mathcal{L}$FSQL. We will continue to talk about what the main functions in the implementations are. This chapter will also contain the discussions on what the datatypes in the implementation are representing $\mathcal{L}$FSQL.

5.1 The Haskell and the Parsec Library

Haskell is a functional programming language which sees programming as functions that accept inputs and produce outputs [19, 27]. Due to its pure functional programming nature, it is easy to produce robust and sophisticated programs using very simple code. Haskell is a strongly typed language. In other words, a data used in Haskell must correspond to its appropriate type. Haskell also uses the lazy evaluation technique. It passes all inputs without evaluating them right away. It starts evaluating them only when they are needed in the execution process. We used Haskell for this work because of its coding simplification and strong type system. In addition, there are certain libraries that are available for Haskell that were very suitable for our implementation.

One of the libraries important to our work is the parser combinator library Parsec. It provides some small parsing functions and operators to construct more sophisticated parsers [22]. It has a lot of modules such as `ParsecCombinator`, `ParsecExpr` and

`ParsecToken`. To parse a structured input such as $\mathcal{L}$FSQL statements, the input is divided into meaningful items called tokens. Based on the given description of the tokens in the parser, the analyzer scans the given statements, identifies these tokens and establishes relations between the tokens [17] . Parsec handles both parsing and lexical analysis [22]. Some useful benefits of Parsec are the alternative function (<|>), the look ahead ability, and a way to customize your own error messages. Parsec is well documented with references which makes it easy to understand and use [16].

The parser function in the Parsec library is what we use to execute the parsers. We supply the parser function with the parser, the input and name of the input for error reporting [16]. The parser function reports either a successful parsing or an error message if the parsing fails.

The module `ParsecCombinator` in the Parsec library is a fast combinator. In other words, a fast predictive parser. Some of the combinators in the library are polymorphic parsers e.g. `many`, `many1`, `skipMany`, and so on. In the Parsec combinator module, we can combine simple functions to develop an advanced parser. These simple functions and parsers in the `ParseCombinator` module were combined to build this $\mathcal{L}$FSQL parser.

Parsec has a standard module called `ParsecExpr` which provides a simple but advanced way of handling expressions. It is an extension which helps to parse expression grammars [16]. The `buildExpressionParser` function takes the basic expression, the operators table as arguments and returns the desired output. The `buildExpressionParser` is very essential in the developing of the **WHERE** clause of the $\mathcal{L}$FSQL parser. Using `buildExpressionParser`, it is easy to specify whether an operator is an infix, a postfix or a prefix as well as the associativity of the infix whether is left or right associative. One can also set the precedence specification on the operators. In other words, which operator is to be executes first in an expression [16].

For example, in the $\mathcal{L}$FSQL parser, we use the logical connectors **OR** and **AND** to combine two or more basic conditions in the **WHERE** clause. We used the `buildExpressionParser` to handle the **WHERE** clause involving logical connectors. We provide explanation in Section 5.3 on how we used the `buildExpressionParser` to build and parse compound conditions in the $\mathcal{L}$FSQL **WHERE** clause.

The `ParsecToken` module extension in the Parsec helps us to deal with other token such as comments, parenthesis and comma. The `makeTokenParser` in the `ParsecToken`

accepts the features of the language as an input and it returns a set of lexical parsers. We used it to obtain tokens such as parenthesis, dot, comma and whitespace to develop the $\mathcal{L}$FSQL parser.

The `ParsecLanguage` module is used to define a common language. It gives the flexibility of specifying the features of your own language. With this module, you can state some features such as comment, identifier, reserved words, etc. In the $\mathcal{L}$FSQL parser, we use this function to define the reserved words, identifiers, comments, reserved operation names, case sensitivity, and others.

The `Control.Applicative` and `Control.Monad` modules provides us with some useful wrapping and binding functions. These functions make it easy and simple to build sophisticated parsers. We used most of these functions to develop the $\mathcal{L}$FSQL parser. The applicative functor (`<*>`) unwraps both sides, performs the needed operations and wraps the results again. The functor (`*>`) unwraps both sides, discards the left side and returns the right side as the result of the operation. The (`<*`) does the opposite of (`*>`).

For example, in the $\mathcal{L}$FSQL delete parser, we made use of these applicative functors as we can see from the code.

```
p\_Delete = Delete <\$> (pReserved "DELETE" *> pReserved "FROM"
            *> p\_TableId) <*> optionMaybe (pReserved "WHERE"
            *> p\_FConditions)
```

To parse the `WHERE` part of the code, the system will unwrap both the `WHERE` and the `p_FConditions`, discard the `WHERE` and return the right which is `p_FConditions`. The same process will occur in the case of the `DELETE` and the `FROM` part in the bracket. In the case of `<*>` it will unwrap the result from `(pReserved "DELETE" *> pReserved "FROM" *> p_TableId)` and `optionMaybe (pReserved "WHERE" *> p_FConditions)` and combine the two together. Lastly, the `<$>` will unwrap the final results from the `FROM` and the `WHERE` operations, combine with the constructor `Delete` and wrap all together as the results from parsing a **DELETE** statement.

5.2 The Datatypes in the Implementation

There are several datatypes used by the $\mathcal{L}$FSQL parser. These datatypes hold the various units which form the $\mathcal{L}$FSQL language. In other words, these datatypes hold

the return values from the various functions in the $\mathcal{L}$FSQL parser. In Haskell, we define a new datatype by stating the name of the datatype after the word data. It is then followed by the equal sign, the name of the constructor and the various existing datatype(s) from which we are building this new datatype [27]. In this subsection, we discuss the various datatypes which hold the internal representation of the various $\mathcal{L}$FSQL statements. We will also provide some of the codes for datatypes in this section to aid the reader's understanding.

In parsing $\mathcal{L}$FSQL, we build higher and more complex datatypes from simple and basic ones. The `Degree` datatype contains the lattice value from the given lattice $\mathcal{L}$. It carries the various signature of the equal or the order classes if they are equal or ordered lattice respectively.

Similarly, the `Value` datatype holds the actual value from the given domain. It holds an integer, a float, a char or a string. The `ExplicitValue` datatype also holds the explicitly listed fuzzy set. In other words, its holds the list of pairs of degree and the actual value. It was built from both `Degree` and `Value` datatypes stated. The `FuzzValue` datatype is made up of `ExplicitValue` and `Labels`. It holds an explicit fuzzy value or a string which is just a label for fuzzy set.

The `AttributeId` datatype holds the string which represents the name of an attributes. The `TableId` also serves a similar purpose as the `AttributeId`. The `TableId` has the name `TableId` as the constructor and a `String` datatype. It holds the name of the table in the $\mathcal{L}$FSQL query statements. A further level datatype built from the `AttributeId` and the `TableId` datatypes is the `QualifiedAttributeId`. The `QualifiedAttributeId` datatype has the same name as its constructor and both `TableId` and `AttributeId` as the existing datatype from which it was built. The code representing `QualifiedAttributeId` is

```
data QualifiedAttributeId = QualifiedAttribute TableId
                            AttributeId deriving Show;
```

The `QalifitedAttributeId` datatype holds an attribute with its table name being attached to distinguish it from other attributes.

The `SelectList` datatype holds the list of attributes to be selected in the **SELECT** clause of an $\mathcal{L}$FSQL **SELECT** statement. It was built from `AttributeId` and `QualifiedAttributeId` as can be seen from the code provided below.

```
data SelectList = SelectAttr (Either [QualifiedAttributeId]
[AttributeId] ) deriving Show
```

The `SelectList` has the select attribute `SelectAttr` as the contructor and uses the monad `Either` to hold either the attributes in the list of `AttributeId` or a list of qualified attributes in the `QualifiedAttribute` datatype.

The `FromClause` datatype also uses the `TableId` datatype. It has `From` as the constructor and holds either one or more table's names in the **FROM** clause.

The `CompareValue` datatype is defined to hold the values or attributes we compare in a basic $\mathcal{L}$FSQL **WHERE** clause. `CompareValue` can hold an attribute or fuzzy Value or qualified attribute.

The fuzzy comparator `FComparator` datatype holds all the alternative fuzzy comparators used in the $\mathcal{L}$FSQL **WHERE** clause. The contructors themselves hold the various kinds of the fuzzy comparators. The `FComparator` can hold any of 12 fuzzy comparators. Below is the code for the `FComparators datatype`.

```
data FComparators = FEqual | NFEqual | FDiff | NFDiff
                  | FGTh | NFGTh | FLTh | NFLTh
                  | FGEqual | NFGEqual | FLEqual
                  | NFLEqual deriving Show
```

The `ThresholdId` datatype has `Thold` as the constructor name and `Degree` to hold the degree of the fulfillment threshold.

The `FCondition` datatype holds the basic $\mathcal{L}$FSQL condition. The `FCondition` and its constructor has the same name with `CompareValue`, `FComparator`, `CompareValue` and optional `ThresholdId` as existing datatypes used to build the `FCondition` datatype. They hold the various elements in the basic $\mathcal{L}$FSQL condition. Below is the code for the `FCondition` datatype.

```
data FCondition = FCondition CompareValue FComparators
                  CompareValue (Maybe Threshold) deriving
                  Show
```

The `LogicalCondi` datatype holds the various compound or complex fuzzy conditions.

The `LogicalCondi` can hold basic conditions or compound conditions connected by the logical connectors. It has the `And` as the constructor for a compound condition using **AND** as the connector. Similarly, it has `Or` as the constructor for a compound condition involving **OR** as the connecting logical connector. In the case of basic $\mathcal{L}$FSQL condition and subquery, it uses `Basic` and `Subquery` respectively. Below is the code for the `LogicalCondi` datatype.

```
data LogicalCondi = And LogicalCondi LogicalCondi
                  | Or LogicalCondi LogicalCondi
                  | Basic FCondition
                  | Subquery LFSqlState  deriving Show
```

The main datatype in the $\mathcal{L}$FSQL parser is `LFSqlState`. This datatype is the highest datatype which holds the **SELECT**, **DELETE**, **CREATE** and **INSERT** $\mathcal{L}$FSQL elements. The `LFSqlState` datatype has `Select` as a constructor with `SelectList`, `TableId` and `Maybe LogicalCondi` as parameters storing the elements of a **SELECT** statement. The other three alternatives are defined similarly using the constructor Delete, Insert and Create. The code for the `LFSqlState` datatype is:

```
data LFSqlState = Select SelectList [TableId] (Maybe
                    LogicalCondi)
                | Delete TableId  (Maybe LogicalCondi)
                | Insert TableId ([AttributeId]) ([FuzzyValue])
                | Create TableId [(AttributeId, AttributeType)]
                deriving Show
```

5.3 The Main Functions of the Implementation

The functions in the $\mathcal{L}$FSQL parser are the programs which extract the tokens in an $\mathcal{L}$FSQL statements and put them in their respective datatypes. Similarly to the datatypes, we build complex and advanced functions from simple and basic functions. Each datatype in the $\mathcal{L}$FSQL parser has a corresponding function to extract the tokens it stores from the $\mathcal{L}$FSQL statements. Before we discuss the main functions `p_LSqlState`, `p_Select`, `p_Insert` and `p_Delete`, we will discuss some of the basic functions upon which we built these complex functions. We will provide the code for some of these functions together with their datatypes to facilitate easy understanding

of the discussions of these functions.

The function which parses a lattice value is the `p_Degree`. In the case of the `p_ExplicitValue`, it parses an explicitly given $\mathcal{L}$-fuzzy set. The `p_FuzzyValue` parses either a label or an explicit $\mathcal{L}$-fuzzy set.

The `p_Attribute` function returns an `AttributeId` datatype. It extracts the attribute part of the $\mathcal{L}$FSQL statement and stores in the `AttributeId`. It uses the `pIdentfier` token or function to identify and extract attributes from the statements. The `p_TableId` works like the `p_AttributeId`. It extracts the name of a table which is a string in an $\mathcal{L}$FSQL statements and returns it as `TableId` datatype value. A higher level function is the `p_QualifiedAttributeId`. This function parses qualified attribute and returns a value of the type `QualifiedAttributeId`. It combines the functions from `p_TableId` and `p_AttributeId` as we can see from the code.

```
data QualifiedAttributeId = QualifiedAttribute TableId
                            AttributeId deriving Show;
p_QualifiedAttributeId :: Parser QualifiedAttributeId
p_QualifiedAttributeId = QualifiedAttribute <$> p_TableId
                         <*>(pDot *> p_AttributeId)
```

It accepts statements with a table name dot attribute name, extracts the table name, the attribute name and store them in their respective datatypes.

The `p_SelectList` returns values of the type `SelectList`. It is the function which works on the **SELECT** clause in the $\mathcal{L}$FSQL **SELECT** statement. It was built on `pCommaSep`, `p_QualifiedAttributeId` and `p_AttributeId`. It uses the `pCommaSep` combinator to identify a list of attributes or qualified attributes seperated by comma and returns them as a list of either `AttributeId` or `QualifiedAttributeId` datatype respectively. The `SelectList` datatype uses the monad `Either` to return the list of `Left QualifiedAttributeId` or `Right AttributeId` datatype.

The `p_FromClause` function returns one or more `TableId` datatype value(s). The `p_FromClause` function uses the `p_TableId` function to parse a table name in the **FROM** clause.

The `p_FComparators` function parses the $\mathcal{L}$-fuzzy comparators. It uses the `try` function in the Parsec to try the various $\mathcal{L}$-fuzzy comparators until it parses the appropriate comparator. For example, from the `p_FComparator` code provided, the `p_FComparator` will first try to parse the given comparator if it is **FEQ** or **F**=. If

it does not succeed, it will try the rest in sequential order until it parses the correct comparator. It returns an error if none of the comparators matches the given comparator.

```
data FComparators = FEqual | NFEqual | FDiff | NFDiff | FGTh
                  | NFGTh | FLTh | NFLTh | FGEqual | NFGEqual
                  | FLEqual | NFLEqual deriving Show

p_FComparators =try(FEqual <$ (pReservedOp "FEQ" <|>
                 pReservedOp "F="))
                <|> (NFEqual <$ (pReservedOp "NFEQ" <|>
                 pReservedOp "NF="))
                <|>(FDiff <$ (pReservedOp "FDIFF" <|>
                 pReservedOp "F<>" <|> pReservedOp "F!"))
                <|>(NFDiff <$ (pReservedOp "NFDIFF" <|>
                 pReservedOp "NF<>" <|> pReservedOp "NF!"))
                <|>(FGTh <$ (pReservedOp "FGT" <|>
                 pReservedOp "F>"))
                <|>(NFGTh <$ (pReservedOp "NFGT" <|>
                 pReservedOp "NF>"))
                <|>(FLTh <$ (pReservedOp "FLT" <|>
                 pReservedOp "F<"))
                <|>(NFLTh <$ (pReservedOp "NFLT" <|>
                 pReservedOp "NF<"))
                <|>(FGEqual <$ (pReservedOp "FGEQ" <|>
                 pReservedOp "F>="))
                <|>(NFGEqual <$ (pReservedOp "NFGEQ" <|>
                 pReservedOp "NF>="))
                <|>(FLEqual <$ (pReservedOp "FLEQ" <|>
                 pReservedOp "F<="))
                <|>(NFLEqual <$ (pReservedOp "NFLEQ" <|>
                 pReservedOp "NF<="))
```

The `p_Threshold` works on the fulfillment threshold part of the statement. It returns a value of the type `threshold`. It uses the `pReservedOp` to extract or identify the word **THOLD** and uses the `p_Degree` to extract the degree of the threshold.

The function which parses the basic $\mathcal{L}$FSQL condition is `p_FConditions`. This func-

tion was built from simple functions such as `p_CompareV`, `p_FComparators` and optionally `p_Threshold` as we can see from the code provided.

```
data FCondition = FCondition CompareValue FComparators
                  CompareValue (Maybe Threshold) deriving Show


p_FCondition :: Parser FCondition
p_FCondition = FCondition <$> p_CompareV  <*>
               p_FComparators <*> p_CompareV <*>
               optionMaybe p_Threshold
```

It parses the various elements in the $\mathcal{L}$FSQL condition and stores it in their respective datatypes.

The `p_FExpression` returns a token of the type `LogicalCondi`. This function is used to identify and extract $\mathcal{L}$FSQL **WHERE** clause involving a logical connectors. We used the `buildExpressionParser` from `ParsecExpr` modulo mentioned in Section 5.1, to identify and parse the needed tokens in a compound $\mathcal{L}$FSQL **WHERE** clause. We provide the basic expression such as `FConditions` and `subqueries` and the operation table `OpElement` which contains the logical connectors. It uses these arguements to build and parse the compound conditions in $\mathcal{L}$FSQL **WHERE** clause. Below is the code used to build the *p_FExpression* parser.

```
 data LogicalCondi = And LogicalCondi LogicalCondi
                   | Or LogicalCondi LogicalCondi
                   | Basic FCondition
                   | Subquery LFSqlState  deriving Show


opElements = [[Infix (And <$ pReservedOp "AND") AssocLeft],
              [Infix (Or <$ pReservedOp "OR") AssocLeft]]


p_FExpression :: Parser LogicalCondi
p_FExpression = buildExpressionParser opElements
                p_FExpression'
                 <?> "expression for the where clause"
           where p_FExpression' = Basic <$> p_FCondition
                                  <|> Subquery <$> p_Select
```

As we can see from the `p_FExpression` code, this parser is a recursive parser. The

<?> is a function in Parsec, We used the <?> to customize our error message.

The p_Create is one of the main functions in the parser. It is the function which identifies and extracts the various elements needed to create a table in the database. It parses **CREATE** $\mathcal{L}$FSQL statement and returns a value of the type LFSqlState. The p_Create uses the pReserved token identifier which identifies the reserved words CREATE and Table in the statement.It also uses p_TableId to identify and parse the name of the table. Lastly, it uses a pCommaSep combinator on p_AttributeId and p_AttributeType to return a list of pairs of attribute and their datatype.

The second main function is the p_Insert. It also returns a value of the type LFSqlState. It is the function which identifies and parses an $\mathcal{L}$FSQL **INSERT** statement. It also uses the pReserved to identify the reserved words INSERT, INTO and VALUES. The p_TableId, pCommaSep, p_AttributeId and the p_FuzzyValue are the basic functions that the p_Insert was built on. They return the name of the table to be inserted into, a list of attributes in the table to hold fuzzy values to be inserted and the fuzzy values themselves.

The third function is the p_Select function. It is the most complex function among all the functions in the $\mathcal{L}$FSQL parser. It combines most of the basic functions. It was built on the p_SelectList, p_FromClause and Maybe p_FExpression functions. These functions parse the list of attribute to select, which table to select from and optionally which condition(s) must be satisfied. The p_Select function combines all the values from these basic functions and returns them altogether as a value of the type LFSqlState. The code for the p_Select function is

```
p_Select :: Parser LFSqlState
p_Select = Select <$> (pReserved "SELECT" *> p_SelectList)
           <*> (pReserved "FROM" *> pCommaSep p_TableId)
           <*> optionMaybe (pReserved "WHERE" *> p_FExpression)
```

The fourth function is the p_Delete. It is the function which parses an $\mathcal{L}$FSQL **DELETE** statement. It uses the pReserved to parse the words DELETE and FROM in the delete statement. It also makes use of the p_TableId to parse the name of the table to delete records from. It uses OptionMaybe combinator to parse a fuzzy condition if it is available or parses nothing if there is no fuzzy condition. The p_Delete function returns a value of the type LFSqlState.

The ultimate function in the $\mathcal{L}$FSQL parser is the p_LFSqlState. This function combines all the other functions in the $\mathcal{L}$FSQL parser. The p_LFSqlState has four

alternative statements to parse as we can see from the code.

```
data LFSqlState = Select SelectList [TableId]
                  (Maybe LogicalCondi)
                 | Delete TableId  (Maybe LogicalCondi)
                 | Insert TableId ([AttributeId]) ([FuzzyValue])
                 | Create TableId [(AttributeId, AttributeType)]
                  deriving Show

p_LFSqlState :: Parser LFSqlState
p_LFSqlState = p_Select <* pSemi <|> p_Delete <* pSemi <|>
               p_Insert <* pSemi <|> p_Create <* pSemi
```

It can parse any of the four $\mathcal{L}$FSQL statements discussed in this thesis. It combines the Parsec alternative combinator with the four main functions discussed in this subsection to parse the **CREATE**, **INSERT**, **SELECT** and **DELETE** statements. This function returns either a create value, insert value, select value or delete value all of the type `LFSqlState`.

Chapter 6

Conclusion

In this thesis, we have presented a generalized relational database i.e. $\mathcal{L}$-fuzzy relational database and its standard language $\mathcal{L}$FSQL. The RDBMS and its SQL are not capable of handling fuzzy values. The FRDBMS and its FSQL are also not capable of handling nonlinear ordered values. Our $\mathcal{L}$-fuzzy relational database and its $\mathcal{L}$FSQL are capable of handling and managing crispness, linear ordered and incomparable values. We achieved this generalization through concepts and properties of $\mathcal{L}$-fuzziness. As shown earlier, every entry into an $\mathcal{L}$-fuzzy relational database is a fuzzy set. From this, we have shown that we can model crisp values as a special case of fuzziness by turning every value to membership degree 0 but only the needed value's membership degree to 1.

As opposed to crispness involving membership degrees either 0 or 1 or fuzzy in the original sense of Zadeh, which involves membership degree of unit interval between 0 and 1 inclusively, $\mathcal{L}$-fuzziness is based on the arbitrary complete lattice. We have shown that lattice makes $\mathcal{L}$-fuzziness more generalized fuzziness which includes both original fuzziness by Zadeh and crispness. The properties and concepts of lattices make it easy to extend your degree of membership and allow incomparable values to exist in its structure. This gives richer notations than the crisp or the fixed unit interval fuzziness. These properties and concepts on which we developed the $\mathcal{L}$-fuzzy relational database and $\mathcal{L}$FSQL have made our relational database more general than the existing ones. It has also paved a way for other fields investigations and future research to be carried based on the semantics of $\mathcal{L}$FSQL

We propose that further research can be done to include other elements such as an update statement. The language can also be expanded to include the having clause, order clause, and the aggregate functions such as sum, average and count. There can also be an expansion to include compatibility degree as was done in FSQL. As stated

above, the semantics of the $\mathcal{L}$FSQL will be an important tool to investigate functional dependencies in the data mining field. Unlike the crisp functional dependencies that requires all values in both sets to satisfy the rules of functional dependencies, $\mathcal{L}$FSQL is more flexible because the sets can satisfy the dependency based on a certain degree. This functional dependencies investigation could me done similar to [23, 31]. Lastly, we propose that an efficient implementation could be done by allowing only certain types of fuzzy sets to be stored. For example, in the implementation of FSQL, only trapezoidal and triangular fuzzy sets were allowed for efficiency.

Bibliography

[1] Birkhoff, G.: Lattice Theory. Amer. Math. Soc. (3rd ed.), 1967.

[2] Chowdhury W.: An Abstract Algebraic Theory of L-Fuzzy Relations for Relational Database, MSc Thesis, Brock University 2015.

[3] Codd, Edgar F.: The Relational Model for Database Management: version 2. Addison-Wesley, 1990.

[4] De Luca, A., and S. Termini.: Algebraic Properties of Fuzzy Sets. Journal of Maths. Anal. and Applicat. 40(2), pp. 373-386, 1972.

[5] Din, Akeel I.: Structured Query Language. Blackwell, 1994.

[6] Galindo, J., Angelica U., and Mario P.: Fuzzy Databases: Modeling, Design, and Implementation. IGI Global, 2006.

[7] Galindo, J., Medina, J. M. and Cubero, J. C.: How to Obtain the Fulfilment Degrees of a Query Using Fuzzy Relational Calculus. In Knowledge Management in Fuzzy Databases, pp. 191-210. Physica, 2000.

[8] Galindo, J., Medina, J. M., Pons, O. and Cubero, J. C.: A Server for Fuzzy SQL Queries. In Flexible Query Answering Systems, pp. 164-174. Springer, 1998.

[9] Goguen, Joseph A.: L-Fuzzy Sets. Journal of Maths. Anal. and Applicat. 18(1), pp. 145-174, 1967.

[10] Grätzer, G. and Davey, B. A.: General Lattice Theory. Springer, 2003.

[11] Galindo J.: New Characteristics in FSQL, a Fuzzy SQL for Fuzzy Databases. WSEAS Trans. on Inform. Sci. and Applicat. 2(2), pp. 161-169, 2005.

[12] https://www.haskell.org

[13] https://hackage.haskell.org/package/parsec

[14] Jackson, E.: L-Fuzzy Relations in Coq. MSc. Thesis, Brock University 2014.

[15] Kawahara, Y., Furusawa H.: Crispness and Representation Theorems in Dedekind Categories. DOI-TR 143, Kyushu University 1997.

[16] Leijen, D.: Parsec, a Fast Combinator Parser.
http://www.cs.uu.nl/daan, 2001.

[17] Levine, J. R., Mason, T., and Brown, D.: Lex & Yacc. O'Reilly, 1992.

[18] Bendre, M., Sun, B., Zhou, X., Zhang, D., Lin, S. Y., Chang, K., and Parameswaran, A. DATA-SPREAD: Unifying Databases and Spreadsheets. VLDB2015 8(12), pp. 2000 – 2003, 2015.

[19] Marlow, S., and Jones, S. P.: The Glasgow Haskell Compiler. Journal of Functional Programming, pp. 393–434, 2002.

[20] Maritz, S. G., Data Management: Managing Data as an Organisational Resource. Acta Commercii 3, pp. 75-84, 2003.

[21] Nation, J. B., Notes on Lattice Theory. Cambridge Stud. in Advanced Math, University of Hawaii, 1998.

[22] O'Sullivan, Bryan, John Goerzen, and Donald Bruce Stewart. Real World Haskell: Code you can Believe in. O'Reilly, 2008.

[23] Ounalli, H., and Jaoua, A.: On Fuzzy Difunctional Relations. Journal of Inform. Sci. 95(3), pp. 219-232, 1996.

[24] Pattis, R. E.: Ebnf: A Notation to Describe Syntax.
http://www.cs.cmu.edu/ pattis/misc/ebnf.pdf.

[25] Schmidt, G. and Ströhlein, T.: Relations and Graphs: Discrete Mathematics for Computer Scientists. Springer, 1993.

[26] Schmidt G.: Relational Mathematics. Encyclopedia of Mathematics and Its Applications, pp. 132, 2011.

[27] Thompson, S.: Haskell: the Craft of Functional Programming. Vol. 2. Addison-Wesley, 1999.

[28] Umano, M. and Fukami, S.: Fuzzy Relational Algebra for Possibility-Distribution-Fuzzy-Relational Model of Fuzzy Data. Journal of Intell. Inform. Syst. 3(1), pp. 7-27, 1994.

[29] Winter, M.: Goguen Categories: A Categorical Approach to L-Fuzzy Relations. Springer, 2007.

[30] Woo, C.: Pseudocomplement. http://www.planetmath.org /pseudocomplement

[31] Yahia, S. B., Ounalli, H. and Jaoua, A: Dynamic Fuzzy Functional Dependency. Journal of Inform. Sci. 119(3), pp. 219-234, 1999.

[32] Zadeh, L. A.: Fuzzy Sets. Information and Control 8(3), pp. 338-353, 1965.